Facultad de Ciencias

Sistema de Control de Asistencia (SAM)

Attendance Management System (SAM)

Trabajo de Fin de Grado
para acceder al
Grado en Ingeniería Informática

Autor: Juan Manuel Lomas Fernández
Director: Mario Aldea Rivas
Co-Director: Miguel Sierra Sánchez
Febrero 2022

Resumen

En este Trabajo Fin de Grado se ha desarrollado un sistema de control de asistencia, como producto para una empresa del sector de las *TIC* (Tecnologías de la Información y la Comunicación) en Cantabria. Sus clientes son mayoritariamente empresas pertenecientes al sector industrial, que abarcan desde pequeños talleres con apenas unos pocos empleados, hasta grandes fábricas con centenares de ellos. Previamente a la implementación de este *software*, en muchas de estas empresas las entradas y salidas del personal se registraban en papel o a través de sistemas cerrados, los cuales no facilitaban el acceso a los datos o la integración con sistemas de terceros. Bajo esta situación, las empresas no podían saber de manera rápida y fiable el grado de absentismo entre sus trabajadores, ni generar los informes necesarios para cumplir con el *Real Decreto de 2019 sobre el control horario laboral*; sin tener que sacrificar un tiempo excesivo recopilando datos en papel y procesando tediosas hojas de cálculo. Esto provocaba no solo la pérdida de tiempo y la pérdida económica asociada a la mala gestión; sino también constantes litigios con los empleados por los desajustes en el registro de sus tiempos de trabajo.

El sistema presentado en este proyecto, proporciona las herramientas necesarias para corregir estos problemas.

Por un lado, se ha desarrollado una aplicación móvil para dispositivos *Android*, que permite fichar a los empleados independientemente del entorno en el que trabajen o de su situación. Se recogen los fichajes en el momento, y se sincronizan en cuanto existe una conexión activa en el dispositivo; lo cual permite no solo fichar a los empleados de una oficina; sino también permite recoger los fichajes de operarios que trabajen fuera de las instalaciones, lo que se conoce como *trabajos en campo*.

Por otro lado, se ha desarrollado una *API REST* para ser ejecutada en un servidor de aplicaciones, y a través de ella los clientes (cómo la aplicación móvil descrita anteriormente, o cualquier otro, como puede ser una aplicación web), pueden conectarse e interactuar con ella a través de la funcionalidad que proporciona. Es la encargada de almacenar y recuperar la información demandada de la base de datos, es decir, que pone a disposición de los clientes la capacidad, no solamente de recoger los fichajes; sino también de realizar todas las gestiones necesarias relacionadas con usuarios, puntos de fichaje, dispositivos u horarios de trabajo, entre otros. Además, permite consultar los datos necesarios para la generación de informes, ya sean de absentismo laboral o de reporte de la jornada efectuada por cada operario. Esto facilita en gran medida el trabajo a llevar a cabo por las empresas, que reducen sus costes de gestión y ayuda a la toma de decisiones.

Palabras clave: Software, Sistema de Control de Asistencia, Industria 4.0, Digitalización de Procesos, Transformación Digital.

Abstract

In this Capstone Project, an attendance control system has been developed as a product for a company in the Information and Communication Technologies (ICT) sector in Cantabria. Its customers are mainly companies in the industrial sector, ranging from small workshops with just a few employees, to large factories with hundreds of them. Until the implementation of this software, in many of these companies, personnel check-ins and check-outs were recorded on paper or through closed systems, which did not facilitate access to data or integration with third-party systems. Under this situation, companies could not quickly and reliably know the degree of absenteeism among their workers, nor generate the necessary reports to comply with the 2019 Royal Decree on working time control; without having to sacrifice excessive time collecting data on paper and processing tedious spreadsheets.This resulted not only in lost time and financial loss associated with mismanagement, but also in constant litigation with employees over mismatches in the recording of their working time.

The system presented in this project provides the necessary tools to correct these problems.

On the one hand, a mobile application has been developed for Android devices, which allows employees to check-in and check-out in regardless of the environment or situation. These check-ins and check-outs are registered instantly and synchronized as soon as there is an available connection on the device; which allows not only to make the check-ins and check-outs in an office; but also to collect them for those operators who work outside the facilities.

On the other hand, a REST API has been developed to be executed inside an application server, and any client (such as the mobile application described above, or any other, such as a web application), can connect and interact with it through the functionality provided by it. It is responsible for storing and retrieving the information requested from the database, i.e., it provides customers with the ability not only to collect the check-ins and check-outs, but also to perform all the necessary steps related to users, control points, devices or work schedules, etc. In addition, it allows to obtain the necessary data for report generation (about absenteeism or about the amount of working time for each operator). This facilitates the work to be made by companies a lot, reduces their management costs and helps them making decisions.

Keywords: Software, Attendance Management System, 4.0 Industry, Process Digitization, Digital Transformation.

Agradecimientos

A mi madre y mi hermana, parte fundamental de mi vida y fuente de apoyo constante.

A mi padre, sin el cual no hubiera empezado a cursar el grado y que ojalá hubiera podido estar para verme acabarlo.

A mis compañeros de universidad y trabajo, por su inestimable ayuda.

A mis profesores de la Universidad de Cantabria, por formarme durante estos años.

A mi tutor, por su consejo y paciencia.

Índice de contenidos

Índice de figuras

Índice de tablas

Introducción

En este documento se detalla el proceso de desarrollo de un sistema cuyo objetivo principal es recoger y consultar los registros de entrada y salida de los empleados de una empresa. El sistema ha sido desarrollado como producto independiente para la compañía SOINCON. SOINCON es una empresa dedicada y especializada en la transformación digital y la digitalización de procesos para el sector industrial. Actualmente, dispone de una *suite* de aplicaciones enfocadas a cada una de las áreas principales de este sector, y que abarcan desde la monitorización de la producción o la gestión de incidencias, hasta la administración de recursos humanos. La implementación del sistema de control de asistencia que aquí se detalla, forma parte de este último módulo, ya que cubre una funcionalidad que actualmente la gestión de recursos humanos de la que se dispone, no contempla.

Para dar forma al proyecto, y siendo este un sistema que aspira a ser parte de un producto y no un *software* a medida, se ha colaborado con tres empresas del sector industrial en la provincia de Cantabria:

- Pequeño taller de calderería, con menos de una veintena de empleados.
- Mediana empresa dedicada a la fabricación de máquinas cableadoras, con una plantilla actual de entre cien y doscientos empleados.
- Una empresa especializada en instalaciones industriales, servicios nucleares, fabricación, mantenimiento y montaje; con más de trescientos empleados repartidos por toda la geografía española.

Estas tres empresas serán referenciadas de ahora en adelante en el presente documento como *"los clientes"*, ya que actúan como una pequeña muestra de lo que serán a futuro los clientes del producto desarrollado.

El objetivo de esta colaboración ha sido principalmente la adquisición de conocimiento sobre los procesos de trabajo actuales, focalizando los puntos en común para la posterior toma de requisitos; y así poder ofrecer un software que resultase útil al mayor número de empresas posible, independientemente de su tamaño y forma de trabajar. En el apartado sobre la metodología utilizada de este documento, se describe más en profundidad cómo ha sido esta colaboración y el impacto que ha tenido en el desarrollo del proyecto.

1.1. Contexto

El control de la jornada laboral es el mecanismo por el que se controla el horario de trabajo llevado a cabo por el empleado, para garantizar el bienestar de este y evitar los fraudes derivados. Cabe destacar que no es algo nuevo, lleva realizándose en nuestro país desde finales del siglo XVI, cuando *Felipe II* instauró el horario de 8 horas para aquellos trabajos que requerían de una exposición prolongada al sol y ha seguido adaptándose y mejorándose desde entonces, beneficiándose de los avances tecnológicos que han surgido desde entonces.

El control de la jornada laboral se puede dividir en dos fases principales: el registro de los fichajes y su posterior revisión y estudio. A día de la redacción de este documento, es uno de los mayores orígenes de disputa y motivo de litigios entre empleados y empresas, debido en la mayor parte de las situaciones, a la ausencia de mecanismos que permitan el registro del *tiempo*

efectivo de trabajo llevado a cabo por el empleado. En otras ocasiones es la falta de integración y la complejidad de los distintos sistemas instalados en las empresas, lo que hace casi imposible poder consultar con exactitud el absentismo laboral. Tampoco ayuda la resistencia al cambio de muchas de las compañías del sector industrial, obsoletas tecnológicamente; en dónde se recoge la información de entradas y salidas del personal en hojas de papel, o en tediosas y desorganizadas hojas de cálculo. Esto lleva mayormente a dos situaciones:

- Operarios de las fábricas que acaban trabajando más tiempo del registrado, o incluso del que debieran por ley trabajar; sin por ello obtener remuneración alguna.
- Empresas que acaban pagando horas extra a empleados por situaciones no productivas, no recogidas en el convenio colectivo.

En la actualidad reciente, concretamente en marzo de 2019, el gobierno aprobó el Real Decreto Ley sobre el control de la jornada laboral, entrando en vigor en mayo de ese mismo año y con el respaldo de la Unión Europea (UE) en su impulso por garantizar los derechos de los trabajadores. En su contenido, se encuentran entre las obligaciones de los empresarios:

> *La empresa garantizará el registro diario de jornada, que deberá incluir el horario concreto de inicio y finalización de la jornada de trabajo de cada persona trabajadora, sin perjuicio de la flexibilidad horaria que se establece en este artículo.*

Además de llevar este registro del horario, la empresa está obligada a guardarlo durante cuatro años y ponerlo a disposición de los empleados y autoridades. Esto implica archivar papel en las empresas, volviendo tedioso y complejo el proceso de consulta y búsqueda.

En la actualidad, muchas si no todas estas empresas que pertenecen al sector industrial, disponen de sistemas y equipos informáticos, capaces de hacer funcionar una pequeña solución software que permita la configuración, el almacenamiento y consulta de estos fichajes. Sumando que a día de hoy todas o casi todas las personas disponen de un teléfono móvil, y que la adquisición de estos dispositivos o tablets es muy asequible económicamente hablando; nos hace pensar que encontrar una solución que permita tanto a empresas como a empleados llevar un registro de la jornada que cumpla con sus expectativas y con las necesidades actuales, es algo muy posible.

1.2. Negocio del cliente

Durante las sesiones con las tres empresas, se ha ido conociendo el negocio de cada una de ellas. Todas ellas necesitan una solución que les permita recoger los fichajes de sus empleados, y no solo eso, sino que también quieren poder distinguir entre aquellos fichajes que son productivos de los que no lo son.

Debido a la variedad en la naturaleza de las tres empresas, se deduce que al menos dos de ellas al tener un tamaño medio-grande, han de poder ubicar los terminales que recojan los fichajes, de alguna forma que les permita saber en todo momento dónde están. La geolocalización para estos casos no es válida, ya que, si tienen por ejemplo las oficinas sobre los vestuarios; no se sabría distinguir en cuál de las dos localizaciones están.

Con relación a esto último, en el caso contrario en el que los empleados van a trabajar a campo, se requiere que se pueda obtener la localización desde la que se ficha, y si es posible, recoger una instantánea del momento del fichaje para dejar constancia. Esta captura, no podrá en ningún caso almacenarse más de 30 días, que es lo que marca la ley. Hay que prestar atención al tamaño

de estas evidencias, ya que en estos casos, varios de los clientes pagan una tarifa de datos para dichos terminales.

También se quiere poder agrupar los empleados de alguna forma, de manera que la administración de estos en un futuro, sea más sencilla.

Además, con vista a sacar los informes de ausencia de los empleados, se tienen que poder contrastar los fichajes realizados por estos, con su jornada laboral. Es por esto, que en esta primera versión se ha de poder asociar un empleado con su jornada. La jornada laboral se define como el conjunto de horarios que tiene asignado un empleado entre dos fechas.

Para finalizar, los dispositivos han de poder conocer cuando se actualizaron por última vez, y qué datos pudieron llegar a actualizar correctamente. Esto es obligatorio, ya que sin esta funcionalidad los terminales no pueden sincronizarse bien.

1.3. Motivación y objetivos

Este proyecto tiene como principal objetivo mejorar el registro de la jornada laboral para las empresas del sector industrial, permitiendo a estas cumplir con lo dispuesto en la ley. Este objetivo se desglosa en los siguientes objetivos concretos:

Objetivos de la aplicación móvil:

- Registrar de manera individual la jornada laboral de los empleados.
- Facilitar el registro de fichajes de los empleados.
- Permitir el registro de fichajes para operarios que trabajen en campo.

Objetivos de la *API REST*:

- Registrar de manera individual la jornada laboral de los empleados.
- Conocer el ajuste del trabajo efectivo realizado por un operario, con su jornada laboral.
- Agilizar el trabajo de gestión por parte del personal de recursos humanos de las empresas.
- Poner a disposición de los empleados el registro de sus fichajes y tiempo de trabajo.
- Poner a disposición de la empresa el conjunto de datos necesarios para consultar el absentismo laboral.

1.4. Organización del documento

En el presente documento se detalla el proceso de desarrollo de una *API REST* y de una aplicación móvil para sistemas Android, ambos escritos en lenguaje Java. Se describen todas las fases involucradas en el desarrollo, desde el análisis inicial de requisitos, hasta la creación de los bocetos de las interfaces de usuario.

El documento se oraganiza de la siguinte manera:

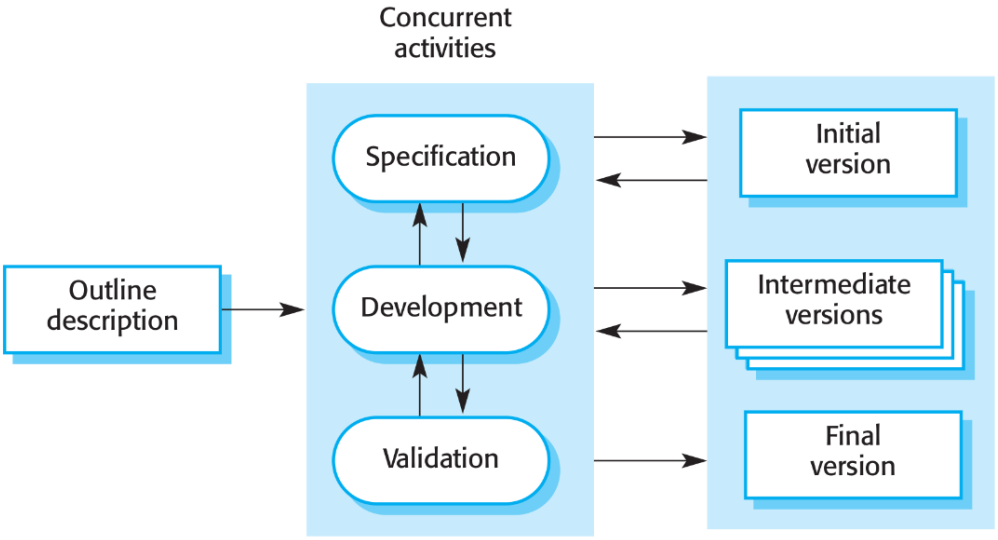
- Capítulo primero: En esta primera parte se pone en contexto al lector, se hace un análisis del negocio de los clientes y se describen los objetivos principales del sistema.
- Capítulo segundo: En la segunda parte, se identifican la planificación, las metodologías y las tecnologías utilizadas.
- Capítulo tercero: En el siguiente apartado, se detallan los requisitos recogidos durante las sesiones iniciales con los clientes. Asimismo, se enumeran los requisitos no funcionales del sistema.
- Capítulo cuarto: En esta sección se hace un estudio exhaustivo de la implementación del cliente Android y de la *API REST*. Se explican las decisiones tomadas durante el desarrollo, desde los patrones de diseño empleados, a la definición de las direcciones para el acceso a los recursos de la *API REST* o el diseño de las interfaces de usuario de la aplicación móvil.
- Capítulo quinto: Después se detallan las pruebas realizadas, entre las que se encuentran las unitarias, de integración y aceptación.
- Capítulo sexto: A continuación, se explica el análisis de calidad llevado a cabo, así cómo del despliegue e implantación del sistema realizados.
- Capítulo séptimo: En este último capítulo se hace una exposición de las conclusiones asociadas a los objetivos del desarrollo. Igualmente, se hace una retrospectiva de la experiencia personal obtenida por el autor durante el proyecto y se describen los trabajos futuros a efectuar.

Metodología y herramientas

2.1. Metodología

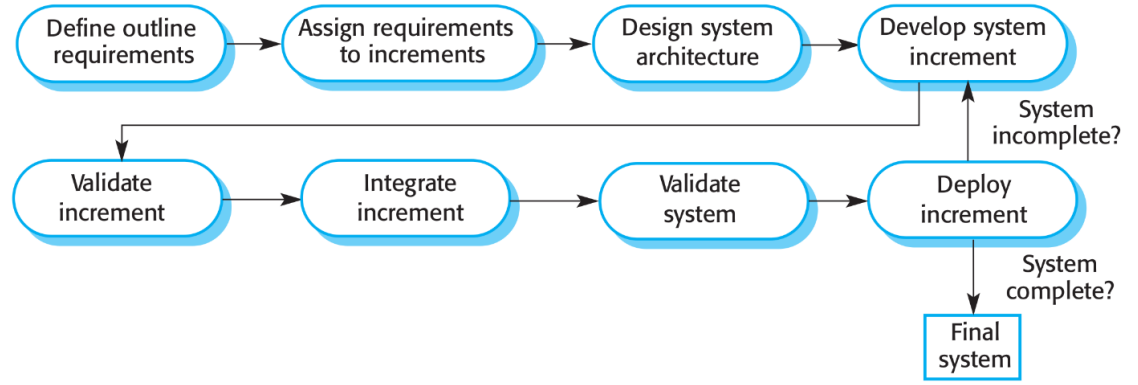
La metodología escogida para el desarrollo del producto ha sido la iterativa incremental. Esta metodología distribuye el trabajo en bloques temporales (*iteraciones*), al final de los cuales se va evolucionando el producto mediante la implementación de nuevos requisitos y la corrección o cambio de los ya existentes (*entrega incremental*). Esta metodología se muestra en la Figura 2.1 y Figura 2.2. Cabe destacar que un aspecto fundamental de esta metodología es la priorización de aquellos requisitos que aportan más valor para los clientes.

Figura 2.1: Desarrollo incremental, [1]



Los motivos principales de esta elección han sido la incertidumbre de los clientes con respecto a lo que espera o necesita y la capacidad que aporta al desarrollador para gestionar sus expectativas. De esta forma se ha podido empezar con requisitos de alto nivel; los cuales a cada iteración se han podido ir refinando, y con lo que los clientes siempre han dispuesto de versiones *usables* desde momentos tempranos del desarrollo.

Figura 2.2: Entrega incremental, [1]



La metodología de trabajo con respecto al versionado del software, se ha realizado utilizando el sistema de control de versiones *Git* [2]. El proyecto siempre ha constado de dos ramas: *develop* y *master*. La primera se ha usado para albergar el trabajo desarrollado de manera continuada; y la segunda es dónde se ha ido volcando a cada iteración, el trabajo ejecutado con su correspondiente etiquetado y entregable o *release*. Cabe destacar que la nomenclatura del versionado empleada, ha seguido las pautas marcadas por estándar definido en el *Versionado Semántico (SemVer) v2.0.0* [3], por el que se sigue la siguiente convención:

> *Dado un número de versión MAYOR.MENOR.PARCHE, se incrementa:*
>
> - *La versión MAYOR cuando se realiza un cambio incompatible con versiones anteriores*
> - *La versión MENOR cuando se añade funcionalidad que es compatible con versiones anteriores*
> - *La versión PARCHE cuando se reparan errores compatibles con versiones anteriores*

2.1.1. Planificación

La planificación del proyecto se ha dividido en cinco fases. Una primera fase para identificar los requisitos generales y de más alto nivel que debía satisfacer la aplicación. A continuación, una segunda fase de desarrollo, que ha constado de seis iteraciones de dos semanas cada una, seguida de una tercera fase de dos semanas en la que se ha implantado el sistema de control de fichajes en la empresa SOINCON, en un entorno controlado y en la que se han hecho las últimas pruebas de integración de extremo a extremo (*End-to-End*), de rendimiento y de usabilidad del sistema por los propios empleados de dicha empresa. Las dos últimas fases corresponden a la implantación de la aplicación en los entornos de preproducción y producción de los clientes.

Durante la primera fase de identificación de requisitos iniciales, cómo al finalizar cada una de las iteraciones del desarrollo; ha sido muy importante la colaboración con las tres empresas citadas con anterioridad en el apartado de *introducción* de esta misma sección. El apoyo recibido ha permitido conocer las necesidades comunes entre negocios muy diferentes, y aquellas partes indispensables para cada una que por la naturaleza o tipo de empresa no se compartían. Cuando finalizaba una iteración del desarrollo, siempre se ha realizado una reunión con los responsables de cada una de las empresas y se les ha enseñado una demostración del desarrollo llevado a cabo durante dicha iteración. De estas reuniones han surgido cambios o modificaciones en los requisitos existentes y han aparecido nuevos requisitos que han sido abordados con las siguientes iteraciones. Gracias a estas sinergias, la empresa SOINCON se ha asegurado que al finalizar el desarrollo, este cumpliera con las expectativas y que el producto fuese atractivo para futuros clientes.

2.2. Tecnologías

En la Tabla 2.1, se recogen las tecnologías y herramientas utilizadas durante la implementación de la *API REST*.

API REST

Tecnología	Descripción
Java	Se ha empleado como lenguaje de programación principal, en su versión JDK 11.
IntelliJ IDEA	Se ha empleado como entorno de desarrollo.
Jackson	Suite de herramientas para el procesado de datos con formato *JSON* en Java.
Spring	*Framework* de alto nivel que permite el desarrollo de aplicaciones de todo tipo en java de forma ágil y rápida. Se compone de módulos que permiten desde la configuración de despliegue rápido del proyecto o el acceso a los datos, hasta la configuración de la seguridad o la programación orientada a aspectos [4].
Hibernate	Framework de persistencia que proporciona una capa de abstracción entre la aplicación Java y la base de datos.
EhCache	Librería de manejo de caché de objetos Java.
MySQL	Base de datos empleada para el desarrollo, que junto al motor *InnoDB*, proporcionan un buen balance de fiabilidad y rendimiento.
Apache Maven	Herramienta principal para la gestión y construcción del proyecto [5] [6].
Apache Tomcat	Contenedor de servlets que se utiliza como servidor para ejecutar aplicaciones web en Java [7] [8].

Tabla 2.1: Tecnologías utilizadas en el desarrollo de la *API REST*

Cliente móvil

En la Tabla 2.2, se recogen las tecnologías y herramientas utilizadas durante la implementación de la *Android*.

Tecnología	Descripción
Java	Se ha empleado como lenguaje de programación principal, en su versión JDK 8.
Android	Sistema operativo para el que se va a desarrollar el cliente móvil. Se ha escogido por su uso extendido y por la facilidad de desarrollo mediante el lenguaje Java y la cantidad de dispositivos existentes en el mercado con dicho sistema.
Android Studio	Se ha empleado como entorno de desarrollo.
Room	Librería de persistencia para dispositivos Android que nos permite trabajar con bases de datos de forma ágil y sencilla
SQLite	Base de datos empleada para el desarrollo. Consume pocos recursos, fiable y funcional.
Retrofit	Cliente *HTTP* para Android y Java. Se utiliza para la comunicación con la *API REST*.
Gradle	Herramienta principal para la gestión y construcción del proyecto.

Tabla 2.2: Tecnologías utilizadas en el desarrollo del cliente móvil

Pruebas

Para las pruebas, se han utilizado las tecnologías y herramientas descritas en la Tabla 2.3.

JUnit	Framework principal para la automatización de pruebas en Java.
Mockito	Framework que nos permite trabajar con mocks a la hora de realizar pruebas.
H2	Base de datos en memoria, empleada para las pruebas de la *API REST*.
SpringTest	Módulo del framework de Spring que nos permite la integración de *JUnit* y *Mockito* con toda la configuración necesaria para su correcto funcionamiento
Selenium	Entorno de pruebas para aplicaciones web, que junto con *Selendroid*, nos permite probar aplicaciones Android nativas.

Tabla 2.3: Tecnologías utilizadas para las pruebas

Además, en ambos proyectos se ha utilizado la librería *Lombok*, la cual permite tener *getters*, *setters* y constructores entre otras cosas, sin la necesidad de escribir nada de código, a salvedad de las anotaciones que se necesiten. Ahorra mucho tiempo, y deja el código limpio.

Análisis de requisitos

A continuación, se describen los requisitos funcionales y no funcionales obtenidos con el cliente, durante las reuniones anteriores al desarrollo del proyecto.

3.1. Definición de conceptos

Antes de explicar qué funciones se esperan o se demandan al sistema a desarrollar, se tienen que definir los conceptos principales que han surgido durante las conversaciones entre la empresa *Soincon* y los clientes, durante la toma de requisitos inicial.

Destacaremos los siguientes términos:

- Empleado: Usuario de la plataforma de fichaje.
- Grupo de empleados: Agrupación lógica de empleados.
- Administrador: Usuario de la aplicación que tiene los permisos necesarios para configurar los dispositivos.
- Punto de control: Ubicación lógica de un dispositivo de fichaje. Se configura por dispositivo y es útil en situaciones dónde las coordenadas del dispositivo no son suficientes para diferenciar la localización de este. Por ejemplo, cuando se tienen dos dispositivos muy juntos (puertas de vestuarios), o cuando las instalaciones tienen varios niveles (oficinas situadas encima de los vestuarios).
- Horario de trabajo: Periodo temporal que delimita un rango de tiempo a trabajar.
- Turno de trabajo: Es un conjunto de horarios de trabajo. Un empleado tiene asociado uno o varios turnos entre dos fechas. Un turno no puede tener horarios de trabajo que se solapen, ni un empleado varios turnos cuyos horarios se solapen.
- Jornada laboral: Es el periodo comprendido por los turnos de trabajos asociados a un empleado.
- Margen de cortesía: Periodo de tiempo que se ignora en los cálculos del tiempo trabajado y en el cumplimiento del horario, a favor del empleado. Es un valor configurable en el sistema. Por ejemplo, si se configura un margen de cortesía de treinta minutos, el turno empieza a las nueve de la mañana y un empleado ficha a las nueve y cuarto; aunque el fichaje se haya realizado un cuarto de hora más tarde del horario estipulado se considera a todos los efectos, que el empleado ha fichado dentro del margen y que, por lo tanto, está cumpliendo su horario.
- Tipo de fichaje: El sistema ha de diferenciar entre fichajes productivos y no productivos. Son valores configurables en el sistema. Por ejemplo, se pueden tener tipos de fichaje para comer (no productivo), para fumar (no productivo), para descansar (productivo) y para entrar al puesto de trabajo (productivo).
- Fichaje: Cada una de las acciones ejecutadas por el empleado con el sistema a través de un dispositivo. Se recoge dónde se ha efectuado, el empleado que ha fichado, la hora y el tipo de fichaje.

- Registro administrativo: Registro por jornada para los empleados. El tiempo de este registro se corresponde al cómputo total del tiempo productivo reflejado en los fichajes hechos por el empleado. Este registro además se utiliza para identificar si el empleado ha cumplido o no el horario, agilizando así las futuras consultas.

3.2. Requisitos funcionales

ID Requisito	Descripción
RF-API-001	El sistema debe permitir dar de alta/editar/eliminar empleados, especificando nombre, apellidos y DNI
RF-API-002	El sistema debe permitir consultar empleados, filtrando por nombre, apellidos, fecha de última modificación o DNI
RF-API-003	El sistema tras dar de alta un nuevo empleado, generará de forma automática un código de acceso para el sistema de control de fichajes
RF-API-004	El sistema debe permitir dar de alta/editar/eliminar puntos de control, especificando nombre y descripción
RF-API-005	El sistema debe permitir consultar puntos de control, filtrando por nombre, fecha de última modificación o descripción
RF-API-006	El sistema debe permitir dar de alta/editar/eliminar turnos de trabajo, especificando nombre y descripción
RF-API-007	El sistema debe permitir consultar turnos de trabajo, filtrando por nombre o descripción
RF-API-008	El sistema debe permitir dar de alta/editar/eliminar horarios de trabajo, especificando nombre, descripción, hora de comienzo y duración
RF-API-009	El sistema debe permitir consultar horarios de trabajo, filtrando por nombre o descripción
RF-API-010	El sistema debe permitir asociar horarios de trabajo a turnos de trabajo
RF-API-011	El sistema debe permitir asociar turnos de trabajo a empleados entre unas fechas. Un empleado puede tener más de un turno asociado dentro de las mismas fechas, siempre que los horarios no se solapen.
RF-API-012	El sistema debe permitir dar de alta/editar/eliminar dispositivos de fichaje, especificando marca, modelo, número de serie, fecha de compra y UUID asociado.
RF-API-013	El sistema debe permitir consultar dispositivos de fichaje, filtrando por marca, modelo, número de serie o UUID asociado
RF-API-014	El sistema debe permitir dar de alta/editar/eliminar tipos de fichaje, especificando nombre, descripción y si el tipo de fichaje se considera productivo o no.
RF-API-015	El sistema debe permitir consultar tipos de fichaje, filtrando por nombre, descripción, fecha de última modificación o si es productivo
RF-API-016	El sistema debe permitir dar de alta/editar/eliminar puntos de control, especificando nombre y descripción
RF-API-017	El sistema debe permitir consultar puntos de control, filtrando por nombre, fecha de última modificación o descripción

Tabla 3.1: Requisitos funcionales para la *API REST 1/2*

RF-API-018	El sistema debe permitir dar de alta/editar/eliminar fichajes, especificando el empleado que realizó el fichaje, el tipo, el punto de control, el dispositivo, el instante de tiempo en el que se llevó a cabo, la geolocalización y si ha sido generado de forma manual o no.
RF-API-019	El sistema debe permitir consultar los fichajes efectuados, filtrando por fechas, nombre del empleado, dni del empleado, tipo de fichaje, punto de control o UUID del dispositivo desde el que se realizó el fichaje.
RF-API-020	El sistema debe permitir dar de alta/consultar las evidencias gráficas (o fotografías) asociadas a un fichaje, especificando la evidencia y el fichaje asociado.
RF-API-021	El sistema debe permitir consultar el tiempo productivo trabajado por cada empleado, día y turno de trabajo. Para ello se llevará a cabo un agregado de los fichajes productivos efectuados por los empleados, por día y turno.
RF-API-022	El sistema debe permitir consultar si los empleados cumplen o no con su horario. Un empleado cumple su horario, si y solo si, el tiempo trabajado durante su jornada se corresponde con su horario configurado, pudiendo incluir en este cálculo un margen de cortesía.
RF-API-023	El sistema debe permitir generar fichajes de forma manual para un empleado.
RF-API-024	El sistema debe permitir completar fichajes incompletos. Se generarán y marcarán como manuales, las parejas de fichajes que no existan y no hayan sido reportadas por el empleado, tomando como referencia el turno y horario de trabajo asignados.
RF-API-025	El sistema debe permitir consultar las ausencias de un empleado. Se contemplarán como ausencias aquellos periodos de tiempo dentro de la jornada de un empleado, en los que no exista tiempo productivo; o en los que el tiempo no productivo supere un valor configurable (margen de cortesía).
RF-API-026	El sistema debe permitir registrar las sincronizaciones de los dispositivos, indicando la fecha de última sincronización y el dispositivo que la llevó a cabo.

Tabla 3.2: Requisitos funcionales para la *API REST 2/2*

ID Requisito	Descripción
RF-APP-001	El sistema debe permitir al administrador acceder al panel de configuración.
RF-APP-002	El sistema debe permitir al administrador consultar y modificar los parámetros de configuración del sistema: ▪ El registro de imágenes para los fichajes. ▪ El registro de la localización para los fichajes. ▪ La configuración del servicio al que conectarse. ▪ La ubicación desde la que se realizó el fichaje. ▪ El tiempo entre sincronizaciones.

Tabla 3.3: Requisitos funcionales para la aplicación móvil *1/2*

ID Requisito	Descripción
RF-APP-003	El sistema debe permitir al empleado acceder al panel de fichaje.
RF-APP-004	El sistema debe permitir al empleado hacer el fichaje. Se guardará el empleado que ha fichado, el instante en el que se hizo el fichaje, el tipo de fichaje, la ubicación y dispositivo desde la que se efectuó. En caso de estar configurado, se guardarán la evidencia gráfica del fichaje o foto; y la geolocalización de este.
RF-APP-005	El sistema debe permitir al empleado consultar sus últimos fichajes.
RF-APP-006	El sistema debe proponer al empleado el siguiente tipo de fichaje a realizar, pero siempre permitirá al usuario revertir este comportamiento de forma manual.
RF-APP-007	El sistema debe sincronizar los datos con la *API REST* configurada. Estos datos son: fichajes, empleados, administradores, dispositivos, puntos de fichaje y tipos de fichaje.
RF-APP-008	El sistema no debe permitir la interacción con el dispositivo, si este no está correctamente configurado.

Tabla 3.4: Requisitos funcionales para la aplicación móvil *2/2*

3.3. Requisitos no funcionales

ID Requisito	Descripción	Tipo
RNF-API-001	La especificación del acceso a los recursos del servicio que constituyen el contrato, ha de ser clara e intuitiva.	Usabilidad
RNF-API-002	Por norma general, toda petición realizada no sobrepasará los 5 segundos de ejecución.	Rendimiento
RNF-API-003	El sistema debe cumplir las disposiciones recogidas en la LOPD respecto al tratamiento de los datos personales de los empleados.	Legal
RNF-APP-004	El servicio REST ha de ser fácilmente escalable y mantenible.	Soporte
RNF-APP-005	Ante un fallo del programa, se ha de recuperar su estado operativo en un periodo de no más de 5 minutos.	Disponibilidad

Tabla 3.5: Requisitos no funcionales para la *API REST*

ID Requisito	Descripción	Tipo
RNF-APP-001	La aplicación ha de visualizarse y funcionar correctamente en cualquier dispositivo Android con una versión 6.0 o superior.	Portabilidad
RNF-APP-002	Un usuario sin experiencia con la aplicación, no tardará más de un minuto en realizar un fichaje de forma correcta.	Usabilidad

Tabla 3.6: Requisitos no funcionales para la aplicación móvil textit1/2

ID Requisito	Descripción	Tipo
RNF-APP-003	La aplicación tendrá acceso restringido a ciertas partes y dicho acceso será configurable de forma sencilla por los administradores.	Seguridad
RNF-APP-004	La aplicación ha de ser fácilmente escalable y mantenible.	Soporte
RNF-APP-005	La aplicación debe permitir la interacción, aunque no disponga de conexión.	Disponibilidad

Tabla 3.7: Requisitos no funcionales para la aplicación móvil textit2/2

Diseño e implementación

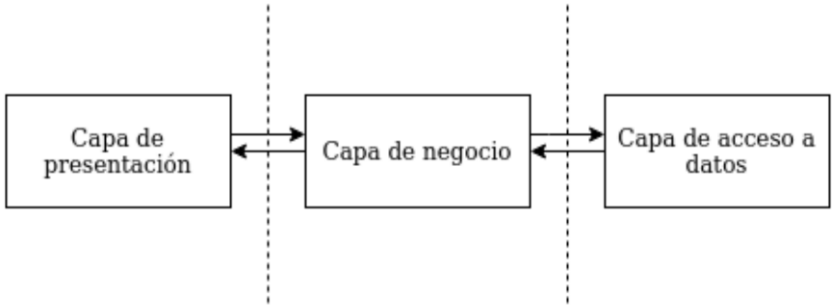
En este apartado se detalla el diseño del sistema, explicando las decisiones tomadas y poniendo ejemplos a cada nivel o capa de la arquitectura utilizada.

4.1. Diseño arquitectónico

Para ambos desarrollos, se ha optado por una arquitectura a tres capas [9], en dónde se tienen separadas la vista de la capa de negocio y del acceso a los datos; y la comunicación entre estas es lineal.

- En la capa de acceso a datos, se tienen normalmente las clases de dominio que representan nuestro negocio, y los *DAOs* (Data Access Objects) que nos permiten operar con las entidades de negocio, esto es, operaciones CRUD, consultas complejas, ejecución de SQL nativas etc.
- En la capa de negocio se encuentran los servicios que se encargan precisamente de eso, de la lógica de negocio, esto es, todas las operaciones de alto nivel con las que se opera tras una petición del usuario y en las que se devuelve el resultado esperado. Además, en esta capa se realiza el control de las transacciones y se protege el dominio de la interacción directa con la capa de presentación.
- En la capa de presentación se encuentra lo que ve el usuario y con lo que se interacciona.

Figura 4.1: Modelo en tres capas



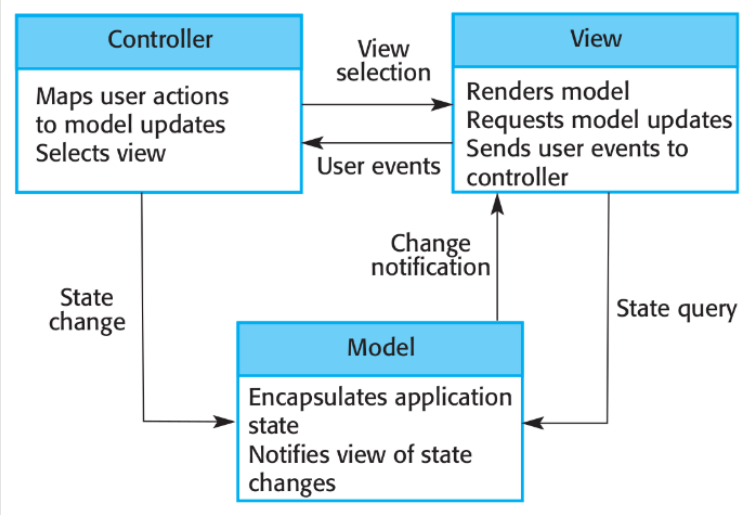
Además, ambos desarrollos están influenciados por el patrón *MVC* (Model View Controller), pero aplicado únicamente a su capa de presentación. Según este patrón de diseño, se dispone de:

- Una la capa de vista, con lo que interacciona el usuario. En el caso de la aplicación Android, se trata de las vistas formadas por los layouts XML. En la *API REST* son los mensajes JSON a través de los que se interacciona con ella.
- En la capa del controlador, está la parte que responde a los eventos y peticiones del usuario a través de la vista, y que invoca dichas peticiones al modelo actualizando la vista posteriormente si es necesario. Esta capa la forman las actividades en la aplicación Android y los controladores web en la *API REST*.
- Por último, está la capa del modelo. En esta capa se llevarían a cabo los cambios en

el modelo (dominio) a través de la lógica de negocio y se actualizaría la vista si fuera necesario. Esta capa no existe como tal en la arquitectura utilizada.

Como ya se dispone de una capa que interacciona con la lógica de negocio, la capa del modelo queda sustituida por dos elementos: por un lado, por la comunicación directa entre el controlador con la capa de negocio existente (los servicios). Por otro lado, como se ha dicho con anterioridad, la capa de negocio tiene como misión proteger al dominio, entonces se necesita una representación de los objetos de negocio que satisfaga las peticiones realizadas por el usuario a través de la vista. Para ello se emplean *DTO's* (Data Transfer Objects), que no son más que las representaciones de información con las que se interacciona entre la capa de presentación y la de negocio; o lo que es lo mismo, entre el controlador y los servicios.

Figura 4.2: Patrón MVC, Ian Sommerville 2016



Como cualquier diseño, existen pros y contras al utilizar esta arquitectura.

Pros:

- Permite tener un desarrollo dividido y bien organizado.
- Facilita la escalabilidad del producto.
- Se puede incrementar el desarrollo o complejidad de una capa sin afectar al resto.
- Permite desarrollar de manera que, si se producen pequeños cambios, no repercutan de manera drástica en el resto del desarrollo.
- Realizar los tests, unitarios y de integración capa a capa, en este tipo de arquitecturas es sencillo.
- Es fácil de reutilizar.

Contras:

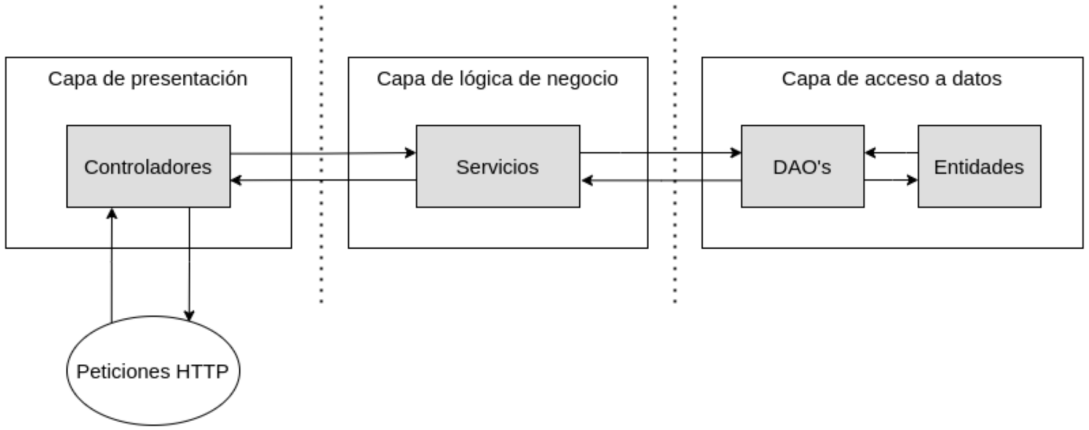
- La necesidad de incluir DTOs implica asumir cierta duplicidad de código, que se acentúa sobre todo, en proyectos pequeños y con poca lógica de negocio.

- Es más compleja de aplicar y requiere más esfuerzo.

4.2. Arquitectura de la API REST

Con lo visto en las secciones anteriores, se puede hacer un esquema con los componentes que se utilizan en cada capa, y que se describen en las siguientes secciones de este documento y en la Figura 4.3.

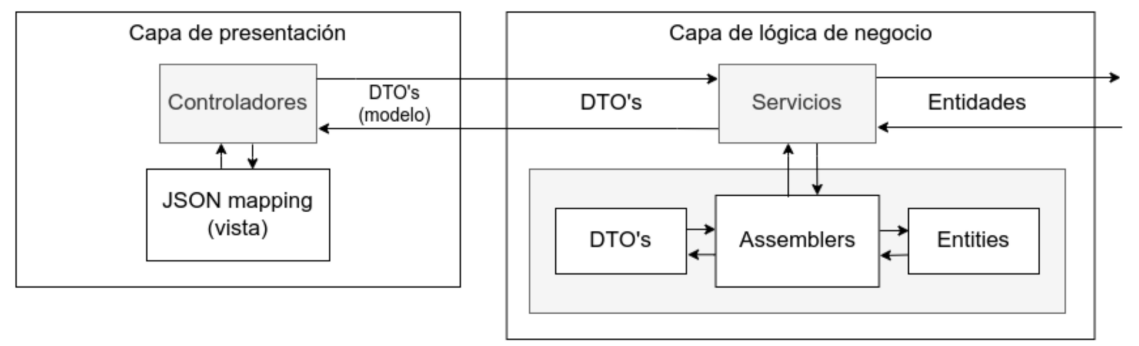
Figura 4.3: Arquitectura API REST 1/2



La capa de acceso a dato, como se puede observar en la Figura 4.3, es la encargada que trabajar en el nivel más cercano a la base de datos. Es la encargada de realizar las operaciones *CRUD* y devolver las entidades de base de datos que se requieran, así como de ejecutar las consultas que se necesiten por parte de la lógica de negocio en los servicios. Es por esto que la comunicación es lineal y directa entre servicios y DAO's.

En la siguiente capa aparecen los servicios, dónde se definen las operaciones de negocio necesarias, agrupándose en las diferentes áreas o bloques de negocio que correspondan. Los servicios se encargan del control de las transacciones en las operaciones, y de traducir el lenguaje más cercano al dominio (a través de las entidades), en las representaciones solicitadas por las operaciones de la capa de presentación (los *DTO's*). De nuevo existe una comunicación directa y lineal entre servicios y controladores.

Figura 4.4: Arquitectura API REST 2/2



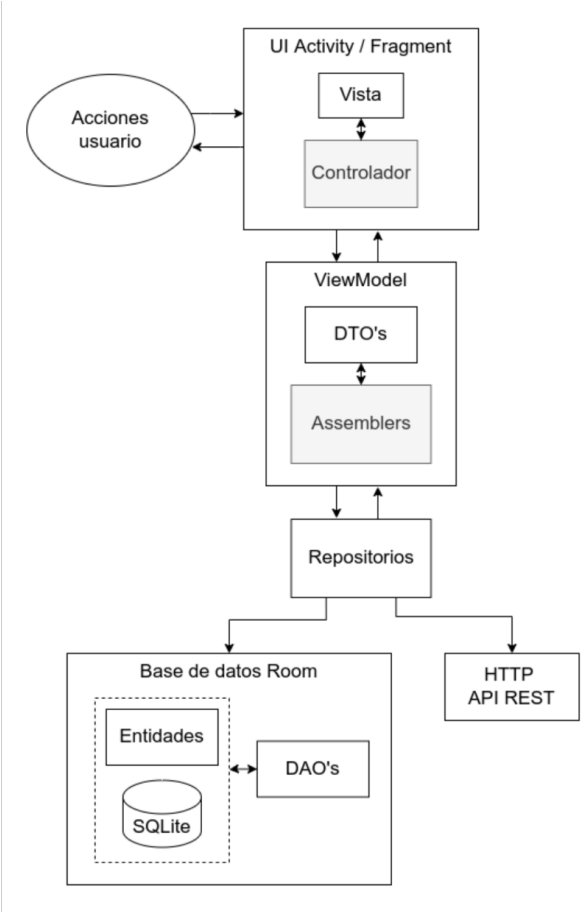
Por último, está la capa de presentación, que en nuestro sistema se refleja en su mayor parte en los controladores. Como se ve en la Figura 4.4, los controladores se encargan de recoger las peticiones *HTTP* realizadas por los clientes, obtener toda la información (cabeceras, operaciones a desempeñar y datos necesarios si los hay), realizar las operaciones llamando a los servicios pertinentes, y devolviendo la representación esperada. Existe una parta transparente para el programador, que es la conversión de esos mensajes en formato *JSON* a los objetos *DTO* con los que se comunican con la capa de servicios. *Spring* se encarga de hacer esa conversión entre *DTO's* y el mensaje recibido o a enviar, utilizando la librería *Jackson*.

Para entender mejor la comunicación entre los diferentes componentes de la arquitectura, en las siguientes secciones de la implementación se muestra un ejemplo con los *Puntos de Control* de la aplicación, dónde se puede observar el flujo de trabajo entre las diferentes capas.

4.3. Arquitectura de la aplicación móvil

En la siguiente Figura 4.5, se muestran la arquitectura y los componentes escogidos para llevarla a cabo en la aplicación móvil.

Figura 4.5: Arquitectura aplicación móvil



En esta arquitectura se aplica el patrón *Repository*, con el cual existe un nivel de abstracción por encima de la capa de obtención de datos, y permite trabajar con diferentes orígenes de forma transparente para las capas superiores. En nuestro caso, esos orígenes de datos serán la pequeña

base de datos que se usa para el almacenamiento temporal de los fichajes y los parámetros de configuración (puntos de control, tipos de fichaje, empleados...) a modo de caché; además de las llamadas a la *API REST* necesarias para la obtención de estos datos. Esta arquitectura no se ha escogido al azar, es la recomendada en el curso avanzado de Android, impartida por [10].

4.4. Seguridad

La seguridad en el desarrollo de la *API REST* se lleva a cabo mediante la utilización de un módulo desarrollado con anterioridad por la empresa SOINCON. El módulo se llama *SNC Security*, está basado en *Spring Security* [11] y se agrega como dependencia al proyecto, teniendo que configurar un fichero de propiedades y decorando los *endpoints* del servicio con los nombres de los permisos requeridos para hacer las correspondientes peticiones.

Figura 4.6: Controlador decorado con los permisos para consultar Puntos de Control

```
/**
 * Returns all existing control points for the client
 *
 * @param clientId the identifier of the client
 * @return all control points
 */
@PreAuthorize("hasAuthority('VIEW_CONTROL_POINTS')")
@GetMapping(path = "/v1/clients/{clientId}/controlPoints")
@Operation(summary = "Returns all existing control points",
        description = "Retrieves every control point for the specified control point id in the path",
        responses = {@ApiResponse(responseCode = "200", description = "Control points retrieved successfully")})
public ResponseEntity<ControlPointCompleteDto> getControlPoints(
        @Parameter(description = "the identifier of the client") @PathVariable Long clientId) {
```

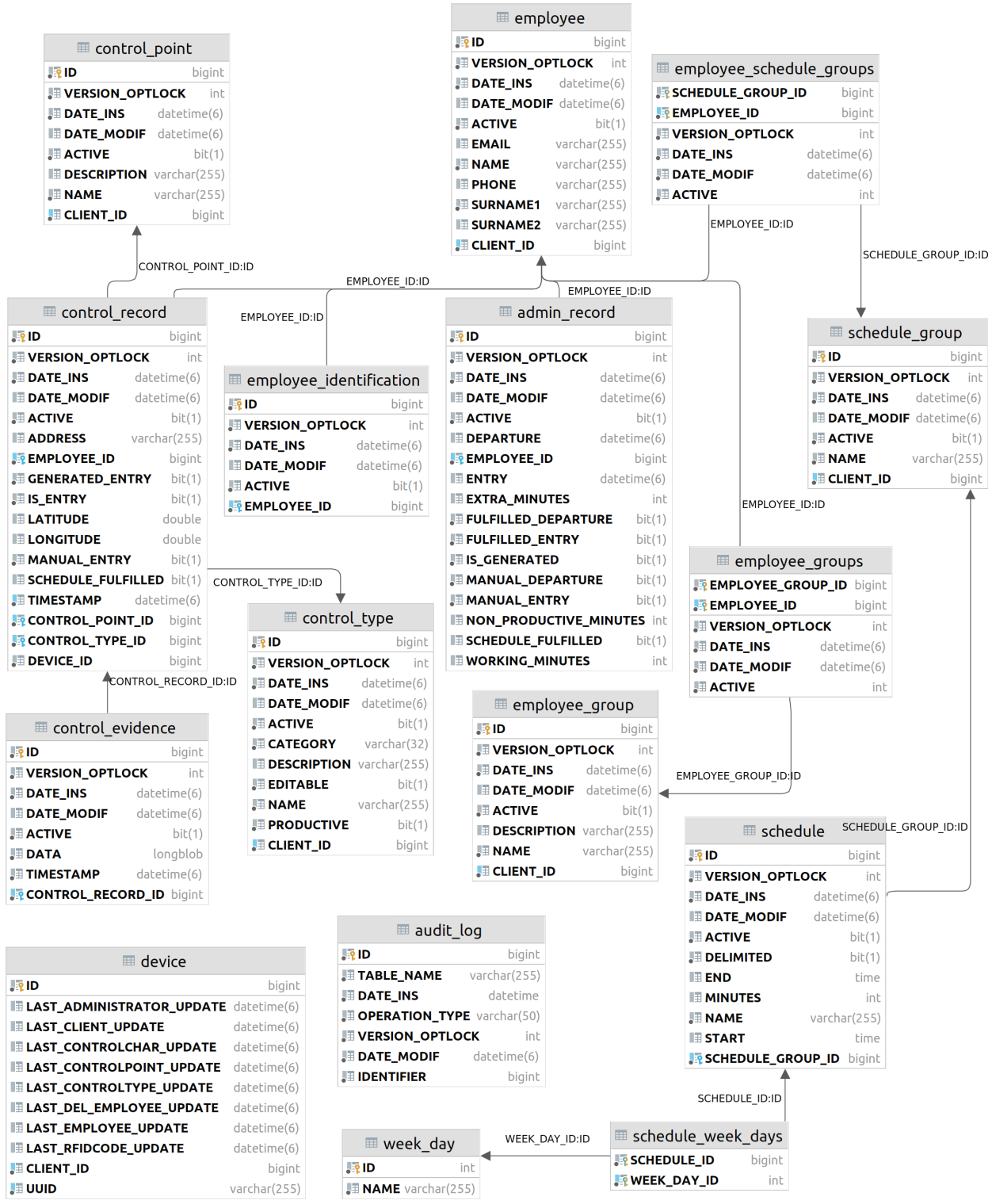
Como se ve en la Figura 4.6, los controladores se decoran con los permisos necesarios para su invocación y de forma transparente al desarrollo, en caso de no realizar la llamada con el token de seguridad o no disponer de los permisos necesarios, se lanzará el correspondiente error 401 o 403. Esto se hace mediante la anotación *@PreAuthorize*, configurándola con los permisos que se deban tener para realizar la solicitud.

Al tratarse de una caja negra, cuyo desarrollo no se contempla en este documento, no se detallará más su funcionalidad.

4.5. Definición del dominio

En el modelo de dominio reside el negocio de nuestro sistema, de una manera que nos permite fácilmente ser trasladado al esquema de base de datos a implementar. Es el punto de partida del desarrollo.

Figura 4.7: Modelo Entidad-Relación



Como se ve en la Figura 4.7, las entidades con más peso dentro del negocio son el *fichaje* y el *empleado*, y a partir de estos, de manera natural surgen el resto de entidades principales y secundarias, cómo son el *los grupos de empleados*, los *tipos de fichaje* o los *puntos de fichaje* y *dispositivos*.

Una vez se tuvieron estas entidades, el resto se fueron identificando de modo paulatino conforme se iba enriqueciendo el esquema aportando más y más información necesaria. Los *registros administrativos* es un ejemplo de una de estas entidades secundarias que de inicio no se hubiera pensado en ella, pero que, tras analizar los requisitos, se dedujo que debía estar para almacenar el agregado de tiempos por jornada y empleado.

4.6. Implementación de la *API REST*

4.6.1. Implementación del modelo y capa de acceso a datos

Para la capa de acceso a datos de la *API REST* (Figura 4.3), se han utilizado: *Spring Data* [12] como framework principal, *Hibernate* como framework de persistencia e implementación de *JPA* (Javax Persistence API), *MySQL* como base de datos principal.

Las entidades son clases JAVA simples (Plain Old Java Object o POJO's), que representan las clases de dominio. Se emplean las anotaciones de *JPA* para decorarlas, el nombre de las tablas y si es el caso, la jerarquía y herencia entre clases de dominio. Además, se definen los tipos de relación con otras clases de dominio, así como el mecanismo de *fetching*.

A continuación, se explican las anotaciones que se pueden observar en la Figura 4.8 del código de la entidad:

@Getter y *@Setter* son anotaciones de *Lombok* para indicar que las propiedades definidas, tendrán los getters y getters correspondientes.

@EqualsAndHashCode es la anotación que indica que las propiedades definidas en la clase, son tenidas en cuenta para la comparación entre objetos.

@NoArgsConstructor es la anotación que indica que se tiene que crear un constructor por defecto sin parámetros.

@Entity es la anotación de *JPA* para indicar que la clase es una entidad, y que representa una tabla de base de datos. Toda entidad ha de tener un constructor por defecto sin argumentos y una clave principal.

@Table es una anotación opcional, pero necesaria si se quiere establecer un nombre para la tabla que no sea la generada por defecto en función del nombre de la clase. Además, permite definir los índices que se necesiten.

@Id es una anotación que indica que la propiedad será la clave principal de la tabla de base de datos.

@GeneratedValue es la anotación que define la estrategia de generación de la clave principal. Esta puede ser de varios tipos, en nuestro caso *IDENTITY*, o lo que es lo mismo, auto-incremental en el caso de MySQL.

Por último, esta *@Column*, la cual indica que la propiedad es una columna de la tabla. Se puede configurar con el nombre, indicando si es puede ser nulo o no, la longitud en caso de ser una campo de texto y mucho más. Esta configuración es tomada en cuenta a la hora de crear o validar el esquema de base de datos por parte de *Hibernate*.

Figura 4.8: Clase de dominio que representa un Punto de Control

```java
/**
 * Entity which represents a control point
 *
 * @author soincon.es
 * @version 202001
 */
@Getter
@Setter
@EqualsAndHashCode(callSuper = true)
@NoArgsConstructor
@Entity
@Table(
        name = "control_point",
        indexes = {
                @Index(name = "UIDX_CONTROL_POINT_NAME", columnList = "CLIENT_ID, NAME", unique = true)
        }
)
public class ControlPoint
extends AbstractEntity<Long> {


    // #############
    // # Constants #
    // #############

    /**
     * Default object description for message composition
     */
    public static final String DEFAULT_DESCRIPTION = "control point";


    // ##############
    // # Properties #
    // ##############

    /**
     * Identifier
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    /**
     * The control point owner identifier
     */
    @Column(name = "CLIENT_ID", nullable = false)
    private Long clientId;

    /**
     * Name of the control point
     */
    @Column(name = "NAME", length = 50, nullable = false)
    private String name;
```

Una vez se han definido todas las entidades, se ha procedido a crear los *DAOs*. En ellos, mediante el uso de la API aportada por el *EntityManager*, se interactúa con las instancias de las entidades a través del contexto de persistencia.

Figura 4.9: DAO para los Puntos de Control (1/2)

```java
/**
 * DAO implementation which let us work with control points
 * when accessing the data
 *
 * @author soincon.es
 * @version 202101
 */
@Repository
@Transactional(propagation = Propagation.MANDATORY)
public class ControlPointDaoImpl
        extends AbstractRepositoryWithFilter<ControlPoint, Long, ControlPointFilter>
        implements IControlPointDao {


    // #############
    // # Constants #
    // #############

    // Field names
    private static final String PROPERTY_ID = "id";
    private static final String PROPERTY_ACTIVE = "active";
    private static final String PROPERTY_CLIENT_ID = "clientId";
    private static final String PROPERTY_NAME = "name";
    private static final String PROPERTY_DESCRIPTION = "description";


    // ################
    // # Constructors #
    // ################

    /**
     * Constructor
     *
     * @param em the entity manager to be used
     */
    public ControlPointDaoImpl(EntityManager em) {

        super(em, ControlPoint.class);
    }


    // ##################
    // # Parent methods #
    // ##################

    @Override
    public List<ControlPoint> findAllByClient(Long clientId) {

        ControlPointFilter filter = new ControlPointFilter();
        filter.setClientId(clientId);

        CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
        CriteriaQuery<ControlPoint> criteriaQuery = criteriaBuilder.createQuery(clazz);
        Root<ControlPoint> root = criteriaQuery.from(clazz);

        List<Predicate> preds = buildPredicates(filter, criteriaBuilder, root);
        List<Order> orderList = buildOrder(criteriaBuilder, root);

        criteriaQuery.select(root);
        criteriaQuery.where(preds.toArray(new Predicate[0]));
        criteriaQuery.orderBy(orderList);
```

Figura 4.10: DAO para los Puntos de Control (2/2)

```
@Override
protected List<Predicate> buildPredicates(
        ControlPointFilter filter,
        CriteriaBuilder criteriaBuilder,
        Root<ControlPoint> root) {

    List<PredicateSource> sources = new LinkedList<>();

    sources.add( new PredicateSource(PredOperationType.EQUAL, root.get(PROPERTY_ID), filter.getId()) );
    sources.add( new PredicateSource(PredOperationType.EQUAL, root.get(PROPERTY_ACTIVE), filter.getActive()) );
    sources.add( new PredicateSource(PredOperationType.EQUAL, root.get(PROPERTY_CLIENT_ID), filter.getClientId()) );

    sources.add( new PredicateSource(PredOperationType.LIKE, root.get(PROPERTY_NAME), filter.getName()) );
    sources.add( new PredicateSource(PredOperationType.LIKE, root.get(PROPERTY_DESCRIPTION), filter.getDescription()) );

    sources.add( new PredicateSource(PredOperationType.IN, root.get(PROPERTY_ID), filter.getIds()) );

    return RepositoryUtil.createPredicates(criteriaBuilder, sources);
}

@Override
protected List<Order> buildOrder(
        CriteriaBuilder criteriaBuilder,
        Root<ControlPoint> root) {

    List<Order> orderList = new LinkedList<>();
    orderList.add(criteriaBuilder.asc(root.get(PROPERTY_NAME)));

    return orderList;
}
```

En las clases correspondientes a los DAO's, se definen los métodos para llevar a cabo las consultas que se necesitan, los filtros a realizar, además de las operaciones *CRUD* básicas.

Todo ello, haciendo uso de la *JPA Criteria API*, la cual nos aporta ventajas:

- Nos permite construir las consultas de manera dinámica y programática.
- Al emplear parámetros, se evitan vulnerabilidades como la inyección *SQL*.

Entre las anotaciones se encuentra alguna interesante, cómo *@Transactional* la cual indica que los métodos definidos en la clase serán transaccionales, y como está configurada con un tipo de propagación obligatoria, en caso de llamar a alguno de estos métodos sin haber una transacción previa, se lanzará un error.

Para aquellas entidades que se vaya a permitir consultar mediante búsquedas, se necesitan también clases de filtro que permitan ejecutarlas. Estas clases son simples *POJO's* en los que se indica cada propiedad que puede intervenir en los filtros. En la Figura 4.9 y en la Figura 4.10 del DAO para Puntos de Control, se puede apreciar su utilización. Se puede ver un ejemplo en la Figura 4.11.

Figura 4.11: Filtro creado para poder hacer búsquedas de Puntos de Control

```java
/**
 * Filter to be used on control point searches
 *
 * @author soincon.es
 * @version 202101
 */
@Getter
@Setter
@EqualsAndHashCode(callSuper = true)
@NoArgsConstructor
public class ControlPointFilter
extends AbstractEntityFilter {


    // ##############
    // # Properties #
    // ##############

    /**
     * Filter for searching by the identifier
     */
    private Long id;

    /**
     * Filter for searching by the identifier (in batch)
     */
    private Collection<Long> ids;

    /**
     * Filter for searching by the name of the control point
     */
    private String name;

    /**
     * Filter for searching by description
     */
    private String description;

}
```

En el caso de la figura anterior, se puede ver que con el filtro se permite realizar búsquedas por identificador (por uno o por varios), por nombre y descripción, y se puede apreciar su uso en la Figura 4.9 y la Figura 4.10, del DAO anterior que contiene el método de búsqueda.

4.6.2. Implementación de la capa de negocio

Una vez se han completado todos los desarrollos de la capa de acceso a los datos, se ha pasado a implementar los servicios.

En la Figura 4.12, aparece parte del servicio asociado a los *Puntos de Control*. Vemos de nuevo la anotación *@Transactional*, debido a que el control de las transacciones se lleva a cabo en esta capa. Así se puede asegurar que la lógica de negocio se ejecute de manera completa.

Figura 4.12: Fragmento del servicio de Puntos de Control

```
@Override
@Transactional(readOnly = true)
public PageDto<ControlPointCompleteDto> getAllByFilter(
        ControlPointFilter filter,
        Pageable pageable) {

    Page<ControlPoint> pageWithEntities = controlPointDao.findAllByFilter(filter, pageable);

    Collection<ControlPointCompleteDto> dtos = pageWithEntities.getContent()
            .stream()
            .map(controlPointAssembler::buildDtoFromEntity)
            .collect( Collectors.toList() );

    return ControllerUtil.buildPageDto(
            dtos,
            pageWithEntities.getTotalElements(),
            pageable.getPageNumber(),
            pageable.getPageSize());
}

@Override
@Transactional(readOnly = true)
public Long countAllByFilter(ControlPointFilter filter) {

    return controlPointDao.countAllByFilter(filter);
}

@Override
@Transactional
public ControlPointCompleteDto create(ControlPointPartialDto dto) {

    ControlPoint entity = controlPointAssembler.buildEntityFromPartialDto(dto);
    entity = controlPointDao.create(entity);

    return controlPointAssembler.buildDtoFromEntity(entity);
}

@Override
@Transactional
public ControlPointCompleteDto createWithUrlValidation(
        ControlPointPartialDto dto,
        Long clientId) {

    DefaultDtoValidator.validate( Arrays.asList(
            new ValidationParam("clientId", clientId, dto.getClientId(), Validation.URL_BODY) ));

    ControlPoint entity = controlPointAssembler.buildEntityFromPartialDto(dto);
    entity = controlPointDao.create(entity);

    return controlPointAssembler.buildDtoFromEntity(entity);
}
```

4.6.3. Implementación del contrato de la *API REST*

En la Figura 4.1, se definen las posibles respuestas por tipo de método invocado en las llamadas que permite la *API REST*.

Método HTTP	Posibles respuestas
GET	200, 400, 401, 403, 404
POST	201, 400, 401, 403, 415
PUT	200, 400, 401, 403, 404, 409, 415
DELETE	204, 400, 401, 403, 404

Tabla 4.1: Respuestas por método *HTTP*

El contrato a seguir para comunicarse con la *API REST*, queda representado en la Tabla 4.2.

Recurso	URL	Métodos HTTP
Tipos de control	/controlTypes	GET, POST
	/controlTypes/search	POST
	/controlTypes/{controlTypeId}	GET, PUT, DELETE
Tipos de control	/controlPoints	GET, POST
	/controlPoints/search	POST
	/controlPoints/{controlPointId}	GET, PUT, DELETE
Dispositivos	/devices	GET, POST
	/devices/search	POST
	/devices/{deviceId}	GET, PUT, DELETE
Fichajes	/controlRecords	GET, POST
	/controlRecords/search	POST
	/controlRecords/{controlRecordId}	GET, PUT, DELETE
Evidencias	/controlRecords/{controlRecordId}/controlEvidences	GET, POST
	/controlRecords/{controlRecordId}/controlEvidences/{controlEvidenceId}	GET, PUT, DELETE
Registros administrativos	/adminRecords	GET, POST
	/adminRecords/search	POST
	/adminRecords/{adminRecordId}	GET, PUT, DELETE
Empleados	/employees	GET, POST
	/employees/search	POST
	/employees/{employeeId}	GET, PUT, DELETE
Grupos de empleados	/employeeGroups	GET, POST
	/employeeGroups/search	POST
	/employeeGroups/{employeeGroupId}	GET, PUT, DELETE
Grupos horarios	/scheduleGroups	GET, POST
	/scheduleGroups/search	POST
	/scheduleGroups/{scheduleGroupId}	GET, PUT, DELETE
Grupos horarios de un empleado	/employees/{employeeId}/scheduleGroups	GET, POST
Cambios registrados	/auditLog	GET

Tabla 4.2: Diseño del contrato de la *API REST*

A la hora de comunicarse cualquier cliente (incluida la aplicación móvil que recoge este desarrollo) con nuestra *API REST*, se hace por medio de los controladores. Estos recogen la información que envíen los clientes en las peticiones (acciones a realizar, cuerpo de los datos, token de seguridad, otras cabeceras...) y permiten la interacción con el sistema, respondiendo de manera oportuna. Con respecto a este apartado, se adjunta cómo ejemplo del desarrollo realizado, parte del controlador de Puntos de Control en la Figura 4.13.

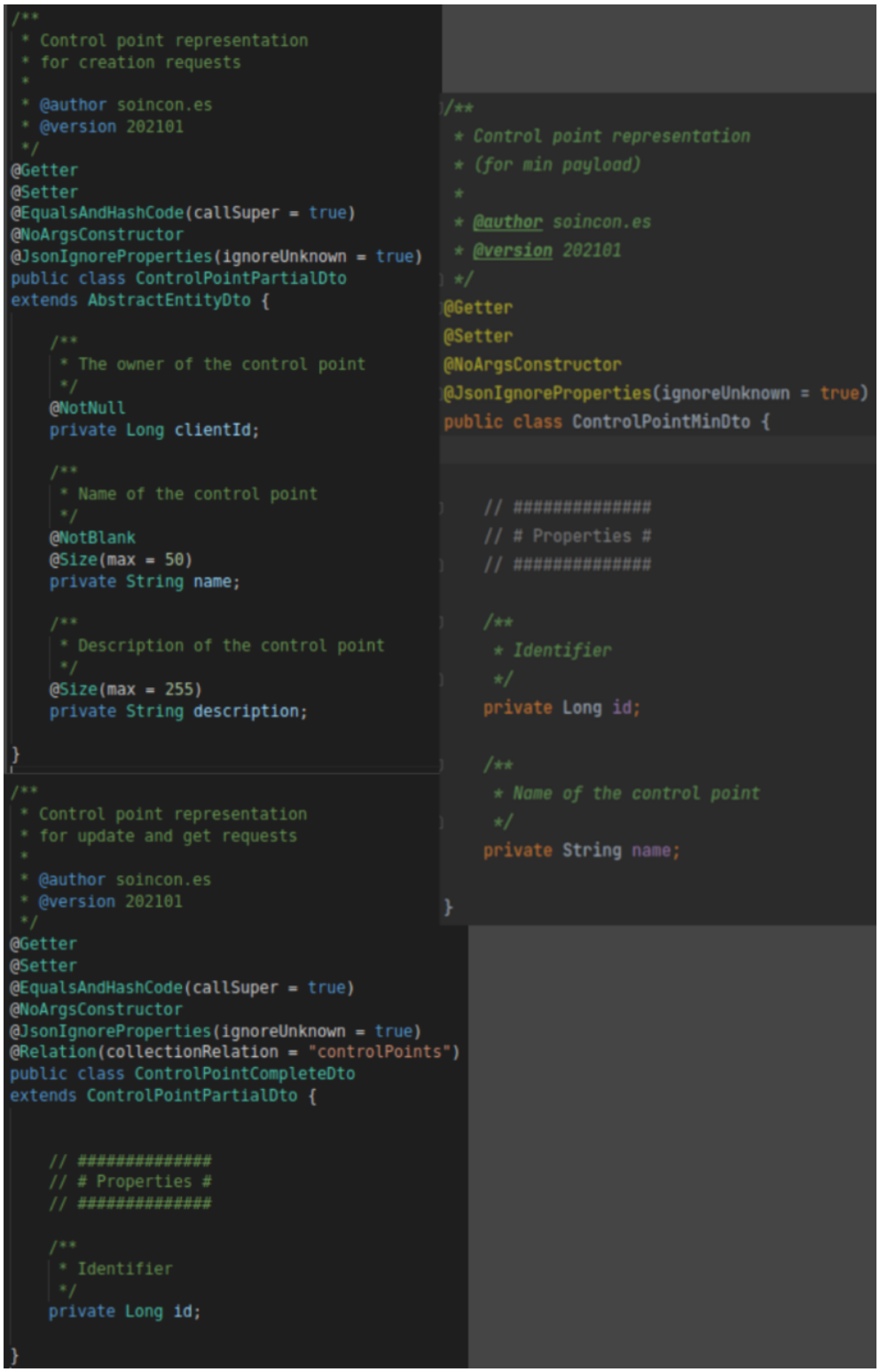
Figura 4.13: Controlador para los Puntos de Control

```java
/**
 * Controller with the contract to be used for
 * control point exchange
 *
 * @author soincon.es
 * @version 202101
 */
@RestController
@RequestMapping(produces = MediaTypes.JSON_VALUE)
@SecurityRequirement(name = "bearer-jwt")
@Validated
public class ControlPointController {


    // ##############
    // # Properties #
    // ##############

    private final IControlPointService controlPointService;


    // ################
    // # Constructors #
    // ################

    /**
     * Constructor that allow us to inject dependencies
     * in a thread-safe way
     *
     * @param controlPointService the service that allow us to work with control points
     */
    public ControlPointController(
            IControlPointService controlPointService) {

        this.controlPointService = controlPointService;
    }


    // ##################
    // # Public methods #
    // ##################

    /**
     * Returns all existing control points for the client
     *
     * @param clientId the identifier of the client
     * @return all control points
     */
    @PreAuthorize("hasAuthority('VIEW_CONTROL_POINTS')")
    @GetMapping(path = "/v1/clients/{clientId}/controlPoints")
    @Operation(summary = "Returns all existing control points",
            description = "Retrieves every control point for the specified control point id in the path",
            responses = {@ApiResponse(responseCode = "200", description = "Control points retrieved successfully")})
    public ResponseEntity<ControlPointCompleteDto> getControlPoints(
            @Parameter(description = "the identifier of the client") @PathVariable Long clientId) {

        return ResponseEntity.ok( controlPointService.getControlPointsByClient(clientId) );
    }

    /**
```

En este tipo de aplicaciones, se tiene que prestar atención al tamaño de los mensajes a intercambiar, de manera general se crean tres DTO's por clase de dominio. Se dice de manera general, ya que habrá de manera excepcional, entidades que quedarán excluidas de esta regla (cómo pueden ser aquellas, fruto de las relaciones N:M entre otras entidades), y aquellas representaciones en

las que se incluyan datos de varias partes del dominio. Por ejemplo, si se necesita consultar los empleados para rellenar una entrada de selección con búsqueda, solo interesan los campos más representativos (DNI, nombre y apellidos) y no traerse los objetos completos que pueden incluir hasta las imágenes de dichos empleados. Para ello se ha dotado a los recursos de la posibilidad de hacer búsquedas, en las que se recibe como cuerpo de la petición un filtro y se puede recibir el resultado de forma reducida o no. Puede verse la aplicación de esta pauta en la Figura 4.14.

Figura 4.14: DTO's que representan un Punto de Control



```java
/**
 * Control point representation
 * for creation requests
 *
 * @author soincon.es
 * @version 202101
 */
@Getter
@Setter
@EqualsAndHashCode(callSuper = true)
@NoArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
public class ControlPointPartialDto
extends AbstractEntityDto {

    /**
     * The owner of the control point
     */
    @NotNull
    private Long clientId;

    /**
     * Name of the control point
     */
    @NotBlank
    @Size(max = 50)
    private String name;

    /**
     * Description of the control point
     */
    @Size(max = 255)
    private String description;

}
```

```java
/**
 * Control point representation
 * for update and get requests
 *
 * @author soincon.es
 * @version 202101
 */
@Getter
@Setter
@EqualsAndHashCode(callSuper = true)
@NoArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
@Relation(collectionRelation = "controlPoints")
public class ControlPointCompleteDto
extends ControlPointPartialDto {


    // ##############
    // # Properties #
    // ##############

    /**
     * Identifier
     */
    private Long id;

}
```

```java
/**
 * Control point representation
 * (for min payload)
 *
 * @author soincon.es
 * @version 202101
 */
@Getter
@Setter
@NoArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
public class ControlPointMinDto {


    // ##############
    // # Properties #
    // ##############

    /**
     * Identifier
     */
    private Long id;

    /**
     * Name of the control point
     */
    private String name;

}
```

Para convertir las entidades a sus representaciones, como parte del patrón *DTO Assembler*, se utilizan clases cuya única responsabilidad será realizar dicha conversión. En la Figura 4.15 se puede ver el ejemplo del *Assembler* para los Puntos de Control.

Figura 4.15: Assembler para los Puntos de Control

```java
/**
 * Assembler for converting control point entities
 * from and to their representations
 *
 * @author soincon.es
 * @version 202101
 */
@Component
public class ControlPointAssembler
extends AbstractAssembler<ControlPointCompleteDto, ControlPointMinDto, ControlPoint> {


    // ################
    // # Constructors #
    // ################

    /**
     * Constructor
     */
    public ControlPointAssembler() {

        super(ControlPointCompleteDto.class);
    }


    // ##################
    // # Parent methods #
    // ##################

    @Override
    public ControlPointCompleteDto buildDtoFromEntity(ControlPoint entity) {

        // Preconditions
        Assert.notNull(entity, "the entity must exist");

        String[] elementsToExclude = ArrayUtils.addAll(AssemblerUtil.ENTITY_DTO_FIELDS,
                "controls");

        ControlPointCompleteDto dto = (ControlPointCompleteDto) DefaultAssembler.buildDtoFromEntity(entity, new ControlPointCompleteDto());
        BeanUtils.copyProperties(entity, dto, elementsToExclude);

        return dto;
    }

    @Override
    public ControlPointMinDto buildMinDtoFromDto(ControlPointCompleteDto dto) {

        // Preconditions
        Assert.notNull(dto, "the representation object must exist");

        ControlPointMinDto minDto = new ControlPointMinDto();
        minDto.setId(dto.getId());
        minDto.setName(dto.getName());

        return minDto;
    }


    // ##################
    // # Public methods #
```

Una vez se tiene la representación del objeto con el que se quiere trabajar, solo queda convertirlo al formato *JSON*. Para ello, *Spring* hace uso de la librería *Jackson*, la cual mediante su clase *ObjectMapper* permite la traducción directa de objetos *POJO* a su representación en formato *JSON*, convirtiendo sus atributos en parejas de valores. Todo esto es transparente al desarrollador, pero se considera importante conocer el funcionamiento de lo que se hace.

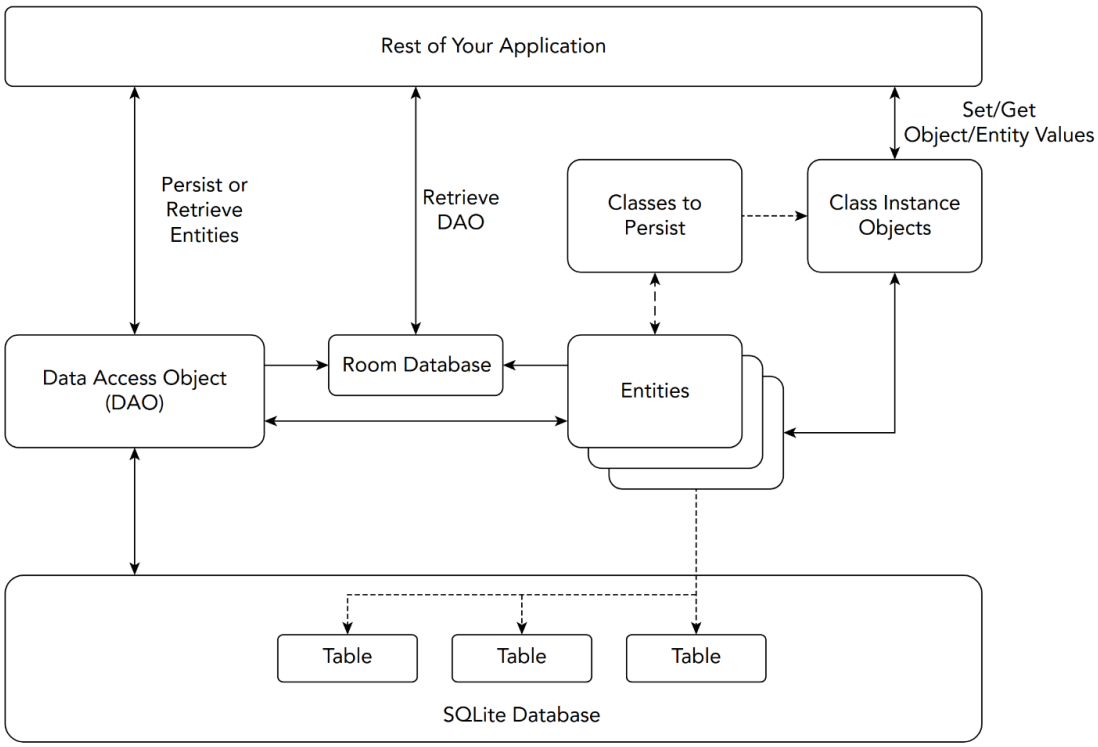
4.7. Implementación del cliente móvil

En los siguientes apartados, se irán detallando los componentes vistos en la arquitectura indicada previamente en la Figura 4.5.

4.7.1. Implementación del modelo y capa de acceso a datos

Para la capa de acceso a datos, se han empleado: *Room* como biblioteca de persistencia y *SQLite* como base de datos principal. El flujo de trabajo con *Room* se puede contemplar en la Figura 4.16.

Figura 4.16: Relación entre el modelo de persistencia, la conexión a base de datos y la aplicación, [13]



Con *Room*, solo hay que centrarse en tres componentes:

- Entidades: Son las clases que representan las tablas y el dominio de nuestra aplicación. No se entrará en más detalles, ya que sus anotaciones e implementación es muy similar a la ya utilizada en el desarrollo de la *API REST*.
- DAO's (Figura 4.17): Las clases que permiten acceder a los datos y consultar o modificar la base de datos. Pese a que la idea es la misma que en el desarrollo anterior, la sintaxis y la forma de trabajar cambian hacia una forma de uso más declarativa.
- Base de datos de Room (RoomDatabase): Esta clase permite el acceso a la conexión de más bajo nivel con *SQLite*, en la que se tiene que especificar los *DAO's* que se va a utilizar.

Figura 4.17: DAO en Room para los Puntos de Control

```java
@Dao
public interface ControlPointDao {

    @Query("SELECT * FROM control_point WHERE id = :id LIMIT 1")
    ControlPoint findById(long id);

    @Query("SELECT * FROM control_point WHERE client_id = :clientId ORDER BY name ASC")
    List<ControlPoint> findAllByClient(long clientId);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insert(ControlPoint entity);

    @Update
    void update(ControlPoint entity);

    @Delete
    void delete(ControlPoint entity);

    @Query("DELETE FROM control_point WHERE id NOT IN (:ids)")
    void deleteAllNotIn(long[] ids);
}
```

4.7.2. Implementación de la capa de negocio

La implementación de la capa de negocio se ha centrado en suministrar los datos necesarios a la vista, y en realizar el resto de funciones en segundo plano transparentes al empleado, entre las que destacan:

- Recoger los fichajes de los empleados, almacenándolos en la base de datos local del dispositivo si no se dispone de conexión. Si el dispositivo dispone del hardware necesario, y está así configurado en el terminal; se recogen con cada fichaje la localización del dispositivo y se hace una fotografía del momento del fichaje.
- Realizar la sincronización con la *API-REST* de los puntos de control, tipos de fichaje, empleados, fichajes y evidencias. Para ello se solicitan los cambios a la tabla de auditoría, efectuados con posterioridad a la fecha de última sincronización. En función de aquello que haya cambiado, se mandan actualizar los datos pendientes y se envían los datos de los fichajes pendientes en cuanto se disponga de una conexión disponible.
- Permitir a los usuarios administradores la configuración del dispositivo. En esta versión inicial se accede mediante una secuencia específica a través del teclado del terminal.

Conexión a una *API REST* desde *Android*

Para la conexión de la aplicación a la *API REST* descrita en el apartado anterior de este documento, se ha hecho uso de una librería que permite abstraerse del uso de la clase *HTTPClient* de Java, la cual necesita más configuración y código.

Este cliente *REST* para *Java* y *Android* se llama *Retrofit*, y permite con una sintaxis muy concisa, establecer la conexión y realizar solicitudes. En la Figura 4.18 se puede observar la implementación del cliente para trabajar con *Tipos de Fichaje*.

Figura 4.18: Definición de las llamadas con *Retrofit* para Tipos de Fichaje

```
public interface ControlTypeApi {

    @GET("/controlTypes")
    Call<List<ControlType>> getControlTypesByClient(@Query("device_id")String deviceId, @Query("force_all")Boolean forceAll);

    @PUT("/controlTypes")
    Call<Boolean> controlPointsOk(@Query("device_id") String deviceId, @Query("last_update") long lastUpdate, @Body long[] controlTypesIDs);
}
```

Obtención de la posición del dispositivo

Para obtener la posición del dispositivo, se utilizan las clases del paquete *android.location*, las cuales nos permite comprobar si el terminal dispone de conectividad *GPS* o está conectado a una red de datos. Si alguna de estas dos opciones está disponible, se pueden obtener las coordenadas y conseguir el nombre de la ubicación a través de la clase *Geocoder* del mismo paquete. Es interesante que se permita obtener y comparar las localizaciones obtenidas por su grado de exactitud. En la Figura 4.19, se puede ver cómo se comprueba qué proveedores para la localización del terminal están disponibles.

Figura 4.19: Consulta del estado de los proveedores de coordenadas

```
locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

if (locationManager != null) {

    // getting GPS status
    isGPSEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
    Log.d(CheckActivity.class.getSimpleName(), (isGPSEnabled) ? "GPS_PROVIDER enabled" : "GPS_PROVIDER disabled");

    // getting network status
    isNetworkEnabled = locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER);
    Log.d(CheckActivity.class.getSimpleName(), (isNetworkEnabled) ? "NETWORK_PROVIDER enabled" : "NETWORK_PROVIDER disabled");

    updateWithLastKnownLocation();
}
```

Captura de imágenes a través de la cámara

Para capturar la instantánea del fichaje, una vez se pulsa en la opción correspondiente en la interfaz, el dispositivo intenta obtener el uso del hardware (en este caso la cámara) y realizar la fotografía. Para minimizar el tamaño de los mensajes y el consiguiente gasto de datos del dispositivo, todas las imágenes capturadas tienen una resolución máxima de 320x240 pixels.

Camera2 [14] es la *API* de *Android* que nos permite llevar a cabo esta función y en la Figura 4.20 se muestra cómo se obtiene la imagen, además de configurar ciertos parámetros cómo el tamaño de la captura o la orientación de la cámara.

Figura 4.20: Método de obtención de imagen a través de *Camera2*

```
private void takePicture() throws CameraAccessException {

    if (null == cameraDevice) {

        Log.e(PACKAGE_NAME,  msg: CLASS_NAME + "cameraDevice is null");
        return;
    }

    final ImageReader reader = ImageReader.newInstance(DEFAULT_WIDTH, DEFAULT_HEIGHT, ImageFormat.JPEG, DEFAULT_MAX_IMAGES);
    final List<Surface> outputSurfaces = new ArrayList<>();
    outputSurfaces.add(reader.getSurface());

    final CaptureRequest.Builder captureBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_STILL_CAPTURE);
    captureBuilder.addTarget(reader.getSurface());
    captureBuilder.set(CaptureRequest.CONTROL_MODE, CameraMetadata.CONTROL_MODE_AUTO);
    captureBuilder.set(CaptureRequest.JPEG_ORIENTATION, getOrientation());

    reader.setOnImageAvailableListener(new ImageReaderUtil(reader),  handler: null);

    cameraDevice.createCaptureSession(outputSurfaces, new CameraCaptureSession.StateCallback() {

                @Override
                public void onConfigured(@NonNull CameraCaptureSession session) {

                    try {

                        session.capture(captureBuilder.build(), captureListener,  handler: null);

                    } catch (final CameraAccessException e) {

                        Log.e(PACKAGE_NAME,  msg: CLASS_NAME + " exception occurred while accessing " + currentCameraId, e);
                    }
                }

                @Override
                public void onConfigureFailed(@NonNull CameraCaptureSession session) {
                }
            }
            ,  handler: null);
}
```

Configuración de la aplicación en modo quiosco

Es importante para varios clientes, poder configurar la aplicación móvil en modo quiosco. Esto significa que no se pueda permitir al usuario que interaccione con el terminal, salir del sistema de control de fichajes o utilizar las funciones del dispositivo para abrir cualquier otra aplicación. En la Figura 4.21 se muestra la configuración del fichero *AndroidManifest.xml* dónde se configura esta característica.

Figura 4.21: Configuración de la aplicación en modo quiosco

```
<receiver
    android:name=".kiosk.AdminReceiver"
    android:label="@string/app_name"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>

    <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin_receiver" />
</receiver>
```

Tareas asíncronas

Para muchas de las funciones que se necesitan en la aplicación, se requiere llevar a cabo tareas que corran en segundo plano sin que perjudiquen a la ejecución de la aplicación principal. En *Android*, toda funcionalidad que se ejecute en una actividad, bloquean el hilo principal que se encarga de gestionar la interfaz de usuario.

Creando tareas extendiendo de la clase *AsynkTask* e implementando una interfaz en las actividades, se pueden realizar trabajos en segundo plano y una vez acabados, emplear un método a modo de función callback para actualizar los datos en la actividad que corresponda.

Figura 4.22: Extracto de la tarea asíncrona que obtiene la localización

```
@Override
protected Boolean doInBackground(final Void... params) {

    address = Util.getAddressName(context, location);

    Log.d(PACKAGE_NAME , CLASS_NAME+ "doInBackground: service called successfully - address");

    // La gestion de excepciones se realiza en la funcion de Util,
    // con lo que siempre devolvera algo y desde aqui true
    return Boolean.TRUE;
}

@Override
protected void onPostExecute(Boolean finishedOk) {

    if(this.listener != null) {

        Completable listener = this.listener.get();
        HashMap<String, Object> result = new HashMap<>();
        result.put("source", GetLocalityTask.class.getSimpleName());
        result.put("finishedOk", finishedOk);
        result.put("address", address);

        listener.onTaskCompleted(result);
    }

    return;
}
```

En la Figura 4.22 se pueden ver los dos métodos principales de las tareas asíncronas: *doInBackground*, dónde se ejecuta la tarea, y *onPostExecute* dónde se retoma el control tras ejecutarse la tarea.

Este mecanismo se utiliza en las tareas para obtener la localización o para sincronizar los datos con la *API REST* , entre otras.

Configuración de los permisos

En el archivo de configuración de la aplicación, también se indican los permisos necesarios para acceder al sistema de ficheros, acceder al posicionamiento del terminal o tomar el control de la cámara. En la Figura 4.23 se muestra un extracto con los permisos otorgados en el proyecto.

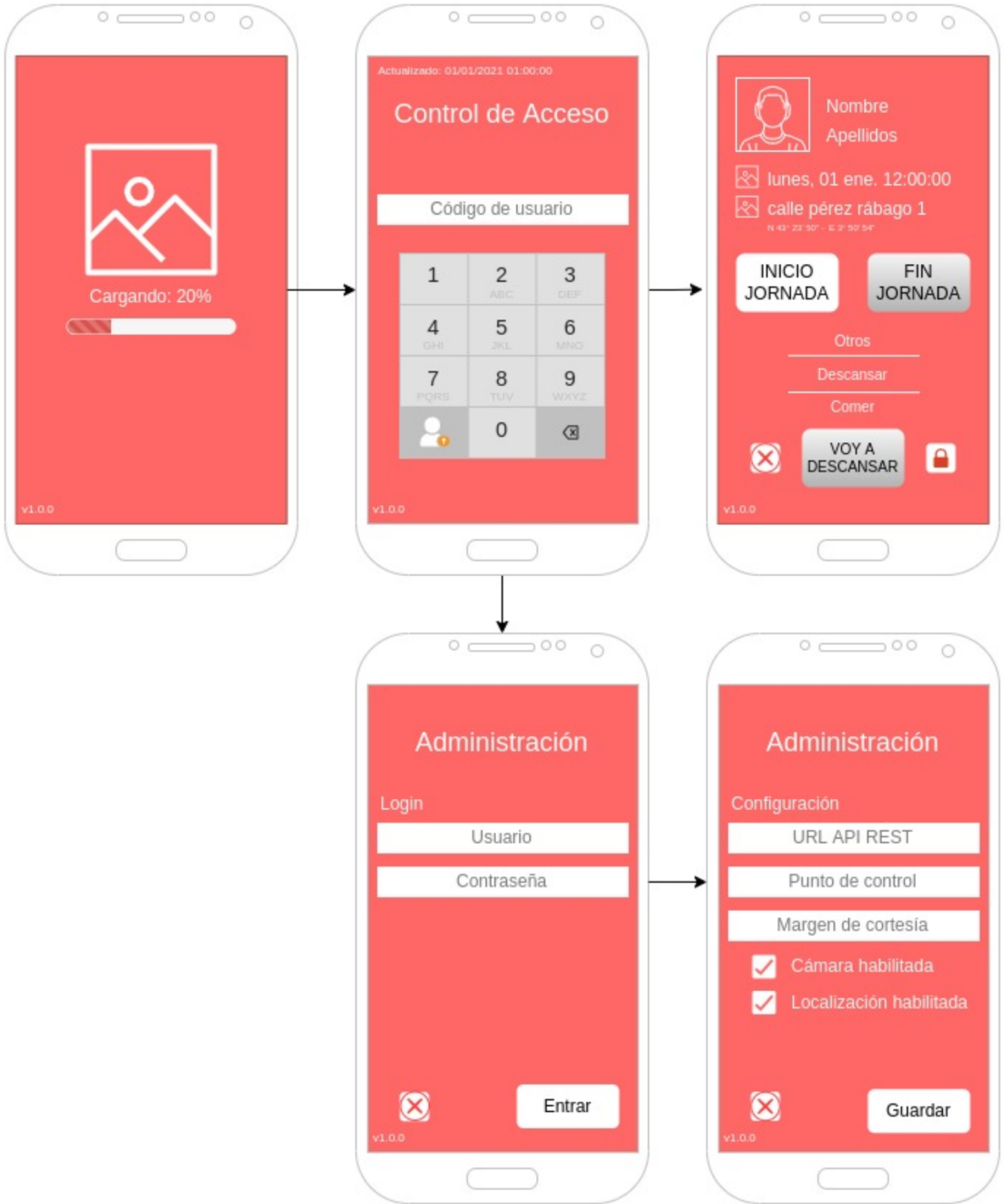
Figura 4.23: Configuración de los permisos de la aplicación

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

4.7.3. Implementación de la capa de presentación

Para el desarrollo del cliente móvil se ha hecho uso de una herramienta que permite la elaboración de los bocetos de las interfaces. La transformación de los bocetos a las interfaces o *layouts* en Android, ha sido casi directa.

Figura 4.24: Bocetos de la interfaz de usuario



En la Figura 4.24 se pueden ver los bocetos de la aplicación móvil y la navegación entre ellas.

Pruebas

Según se ha avanzado en la implementación del proyecto, se han ido probando cada uno de los módulos y funcionalidades de los que consta, por separado y en conjunto; de manera que cada funcionalidad estuviera probada antes de pasar a la siguiente. Ha sido un proceso minucioso, sin llegar a lo que sería un desarrollo guiado por pruebas o *Test-Driven Development (TDD)*; pero sí lo suficientemente concienzudo como para conseguir una incidencia muy baja en la aparición de bugs y la necesidad de aplicar refactorizaciones.

A continuación, se describen más en profundidad cada uno de los ámbitos de prueba realizados.

5.1. Pruebas unitarias

Conjunto de pruebas aplicadas a cada parte o módulo del código de forma aislada, para comprobar su correcto funcionamiento en solitario. Se han efectuado sobre todas las partes de las aplicaciones (en cuanto a controladores, servicios, repositorios y clases de utilidad). En estas pruebas, siempre que se requiere de la interacción con componentes de otras capas, se simula su comportamiento mediante *Mocks*.

Durante estas pruebas no solo se ha hecho caso de las entradas y salidas de los módulos funcionales, sino que también se ha realizado una correcta cobertura de sentencias en el código a probar.

Figura 5.1: Ejemplo prueba unitaria DAO

```
@Test
void testCreateWithoutName() {

    ControlPoint controlPoint = new ControlPoint();
    controlPoint.setClientId(DEFAULT_CLIENT_ID);

    assertThrows(PersistenceException.class,
            () -> dao.create(controlPoint),
    "A new control point should not be created with a null name property");
}
```

En la Figura 5.1 se puede ver cómo se verifica que, en caso de crear un punto de control sin nombre (el cual es un campo obligatorio), se lanza la correspondiente excepción.

Figura 5.2: Ejemplo prueba unitaria servicio

```
@Test
void testGetOne() {

    ControlPoint entity = new ControlPoint();
    entity.setClientId(DEFAULT_CLIENT_ID);
    entity.setName(DEFAULT_NAME);

    ControlPointPartialDto dto = new ControlPointPartialDto();
    dto.setClientId(DEFAULT_CLIENT_ID);
    dto.setName(DEFAULT_NAME);

    Mockito.lenient().when(repository.findOne(DEFAULT_ID))
            .thenReturn(entity);

    Mockito.lenient().when(assembler.buildDtoFromEntity(entity))
            .thenReturn(dto);

    dto = service.getOne(DEFAULT_ID);

    assertEquals(DEFAULT_CLIENT_ID, dto.getClientId(), "The control point should have the given client identifier");
    assertEquals(DEFAULT_NAME, dto.getName(), "The control point should have the given name");
}
```

Para los test unitarios, no existe la comunicación entre capas (que se aborda con los tests de integración). Por ello esta comunicación entre los servicios y los repositorios se simula mediante el uso de *Mocks*, preconfigurando la respuesta. Lo que si se comprueba es que la llamada desde los servicios a la capa de acceso a datos se sa haga, tal y como se observa en la Figura 5.2.

Figura 5.3: Ejemplo prueba unitaria controlador

```
@Test
void testGetControlPoints() throws Exception {

    ControlPointPartialDto dto = new ControlPointPartialDto();
    dto.setClientId(DEFAULT_CLIENT_ID);
    dto.setName(DEFAULT_NAME);

    Collection<ControlPointPartialDto> dtos = new LinkedList<>();
    dtos.add(dto);

    Mockito.lenient().when( service.getControlPoints() )
            .thenReturn(dtos);

    MvcResult result = mockMvc.perform( get( String.format("/controlPoints") ) )
            .andExpect( status().isOk() )
            .andExpect( jsonPath("$[0].name", equalTo(DEFAULT_NAME)) )
            .andReturn();

    dtos = mapper.readValue(result.getResponse().getContentAsString(),
            new TypeReference<>() {});

    assertTrue(dtos.contains(dto), "Controller must return a representation which fits with the given entity");
}
```

En las pruebas unitarias de los controladores (Figura 5.3), se verifican varias cosas:

- Que las direcciones de acceso a los recursos estén bien definidas.
- Que las validaciones del formato de los datos recibidos o enviados sea el correcto.
- Que se hacen las llamadas a los servicios que correspondan (simulando la respuesta).
- Que se devuelven las respuestas correctas en cada caso.

5.2. Pruebas de integración

Las pruebas de integración son las encargadas de verificar el correcto funcionamiento de los diferentes módulos entre sí. En este proyecto se han ejecutado pruebas de integración que incluyen las pruebas *End-to-End (E2E)*, dónde se revisa que, desde la llamada a un controlador, pasando por el servicio, hasta el acceso a los datos funcione correctamente; prestando especial atención no solo a las entradas y salidas esperadas, sino también a otros aspectos cómo el control de las transacciones o el correcto funcionamiento de las cachés.

Figura 5.4: Ejemplo prueba integración

```
@Test
void testGetControlPoint() throws Exception {

    ControlPointCompleteDto dto = new ControlPointCompleteDto();
    dto.setClientId(DEFAULT_CLIENT_ID);
    dto.setName(DEFAULT_NAME);
    dto.setDescription(DEFAULT_NAME);

    dto = controlPointService.create(dto);

    MvcResult result = mockMvc.perform(
                    get( String.format("/v1/clients/%d/controlPoints/%d", DEFAULT_CLIENT_ID, dto.getId()) ))
            .andExpect( status().isOk() )
            .andExpect( jsonPath("$.name", equalTo(DEFAULT_NAME)) )
            .andReturn();

    ObjectMapper mapper = new ObjectMapper();
    mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
    mapper.registerModule(new Jackson2HalModule());

    EntityModel<ControlPointCompleteDto> responseModel = mapper.readValue(
            result.getResponse().getContentAsString(),
            new TypeReference<>() {});

    assertTrue(result.getResponse().getStatus() == HttpStatus.OK.value(), "The response should be 200-OK");
    assertNotNull(responseModel, "The response cannot be null");
    assertEquals(dto.getId(), responseModel.getContent().getId(), "The response should be the same object which was created at the beginning");
}
```

5.3. Pruebas de aceptación

Una vez cubierta toda la funcionalidad mediante pruebas automatizadas, hay que pasar a las pruebas de aceptación. Estas pruebas se han tomado con el cliente y se pueden definir como las pruebas que el software tiene que pasar para determinar si se cumple o no con los requerimientos definidos (Figura 5.5). Se trata de un subconjunto dentro de las pruebas de integración *End-to-End (E2E)*, pero en vez de realizarse de forma automática, se han realizado de forma manual; primero en un entorno controlado dentro de la propia empresa de SOINCON, y luego más tarde en el entorno de preproducción de mano de los clientes.

Figura 5.5: Ejemplo pruebas aceptación

Ref	Prueba	Resultado esperado	ok	ko	Comentarios
1	El empleado se identifica con su código de acceso en el dispositivo	El dispositivo reconoce al empleado, y le muestra la pantalla de fichaje	x		
2	El empleado realiza un fichaje inicial productivo.	El sistema registra el fichaje y notifica al usuario antes de mostrar de nuevo la pantalla de identificación	x		
3	El empleado realiza un segundo fichaje, esta vez no productivo.	El sistema registra el fichaje y notifica al usuario antes de mostrar de nuevo la pantalla de identificación	x		
4	Se desconecta el dispositivo de cualquier red de datos. El empleado realiza un último fichaje, de nuevo productivo.	El sistema registra el fichaje y notifica al usuario antes de mostrar de nuevo la pantalla de identificación	x		
5	Se vuelve a conectar el dispositivo a una red de datos.	El sistema, pasado el tiempo de actualización, se conecta y sincroniza los datos.	x		

5.4. Pruebas de rendimiento

Durante el desarrollo y sobre todo en la fase de implantación en el entorno controlado de la empresa SOINCON, se han hecho pruebas de rendimiento, tanto de la *API REST* cómo del cliente móvil. Estas pruebas han consistido en las siguientes prácticas:

- Se ha comprobado el correcto desempeño de las consultas por parte de la *API REST*, mediante la consulta de los *logs* del framework de persistencia y los *logs* de la base de datos, del impacto de las consultas. Se ha verificado que no se produce el problema de las *N+1 consultas* a la hora de trabajar con *Hibernate*, tal y cómo se recomienda en el libro de [15].
- Se han utilizado herramientas de monitorización para hacer el seguimiento de los consumos de memoria y utilización de CPU de las aplicaciones.
- Se han llevado a cabo pruebas de estrés contra la *API REST*, incluyendo accesos simultáneos para verificar que las políticas de *Bloqueo optimista* empleadas en el desarrollo se han aplicado correctamente.

5.5. Pruebas del sistema

5.5.1. Experiencia de usuario

Como parte de la estrategia para la mejora de la experiencia de usuario de la empresa SOINCON, se ha escogido a un grupo reducido de empleados del cliente a los que se les ha dado acceso de forma remota a la aplicación móvil del sistema de control de asistencia. Al finalizar cada iteración, con la entrega de la versión del software correspondiente, a estos empleados se les ha aportado una lista con operaciones a realizar y preguntas generales sobre la aplicación móvil, en dónde han podido evaluar el grado de dificultad a la hora de interactuar con la aplicación. Se muestra

un ejemplo del documento utilizado en la Figura 5.6.

Este *feedback* se ha utilizado para repasar requisitos existentes y crear nuevos; derivados sobre todo de la distribución y cantidad de elementos en pantalla.

Figura 5.6: Ejemplo checklist experiencia de usuario

Rellene la siguiente hoja después de interaccionar con la aplicación de control de asistencia proporcionada por SOINCON S.L.

Puesto de trabajo	

A continuación, marque una de entre las opciones disponibles (sí o no), en función de su experiencia de uso:

Descripción	sí	no
Considero que existe mucho texto en pantalla y me cuesta leer toda la información		
Los colores de la aplicación me parecen agradables y la combinación no me molesta		
Me cuesta identificar cuando he realizado una acción con la aplicación, no me notifica lo suficiente		
El tamaño de la letra en pantalla es lo suficientemente grande como para leer los textos sin forzar la vista		
Me cuesta navegar por la aplicación, me pierdo entre las opciones disponibles		

A continuación, marque una de entre las opciones disponibles (fácil, normal, difícil), en función de su experiencia de uso:

Acción	fácil	normal	difícil
Identificarse en la aplicación			
Seleccionar el tipo de fichaje a realizar			
Realizar el fichaje			

5.5.2. Seguridad y acceso

Para probar el acceso al cliente móvil, se han generado dos usuarios: un usuario de tipo empleado, y otro de tipo administrador. Para el empleado se ha confirmado que puede iniciar sesión en la aplicación móvil, mediante su código de acceso o a través de la tarjeta *RFID*, y que puede realizar el registro de entrada y salida del trabajo correctamente. Para el administrador se ha comprobado que tiene acceso al apartado de configuración de los dispositivos móviles y que puede modificarlos sin problema.

En el apartado de diseño de esta memoria, se ha descrito el funcionamiento de la seguridad en el backend y no aplica la realización de pruebas. Como el módulo de seguridad SNC-Security es un módulo privado y confidencial, ya desarrollado por la empresa SOINCON y utilizado en otros productos; no se entrará en más detalle sobre él en este documento.

Calidad e implantación

6.1. Análisis de calidad

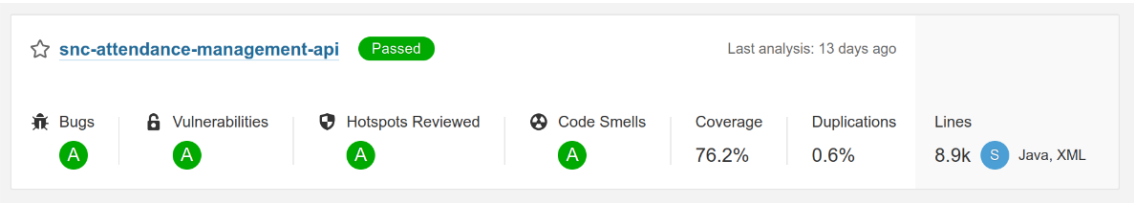
Como medida para afianzar la implementación de este proyecto y satisfacer unos estándares mínimos de calidad en la realización de este, se han llevado a cabo continuos análisis en la calidad del código. Como parte del sistema de integración continua en la empresa SOINCON, con cada cambio en el control de versiones, se llevan a cabo dos operaciones: el paso de pruebas automatizadas y en análisis mediante la herramienta *SonarQube* de la calidad del código desarrollado.

Para medir la calidad, se han configurado en la plataforma una serie de métricas que la empresa considera necesarias para contrastar con el trabajo realizado. En función de estas métricas, *SonarQube* indica si el código analizado es apto o no mediante la función que determina el grado de cumplimiento de las reglas configuradas, también conocida como *Quality Gate*. Nos informa sobre la fiabilidad, mantenibilidad y seguridad del código y nos indica el porcentaje de *deuda técnica* acumulada. En la Figura 6.1 se puede ver el resultado obtenido en el proyecto de la *API REST* con dichas métricas.

El desarrollo del proyecto ha pasado los baremos configurados por la empresa SOINCON, los cuales han ayudado mucho en la identificación de problemas relacionados, entre otros, con:

- Código duplicado.
- Aplicación de estándares de codificación.
- Casos de excesiva *complejidad ciclomática*.
- Casos de excesiva *complejidad cognitiva*.
- Fallos en la documentación y comentarios en el código.

Figura 6.1: Análisis de calidad con *SonarQube*



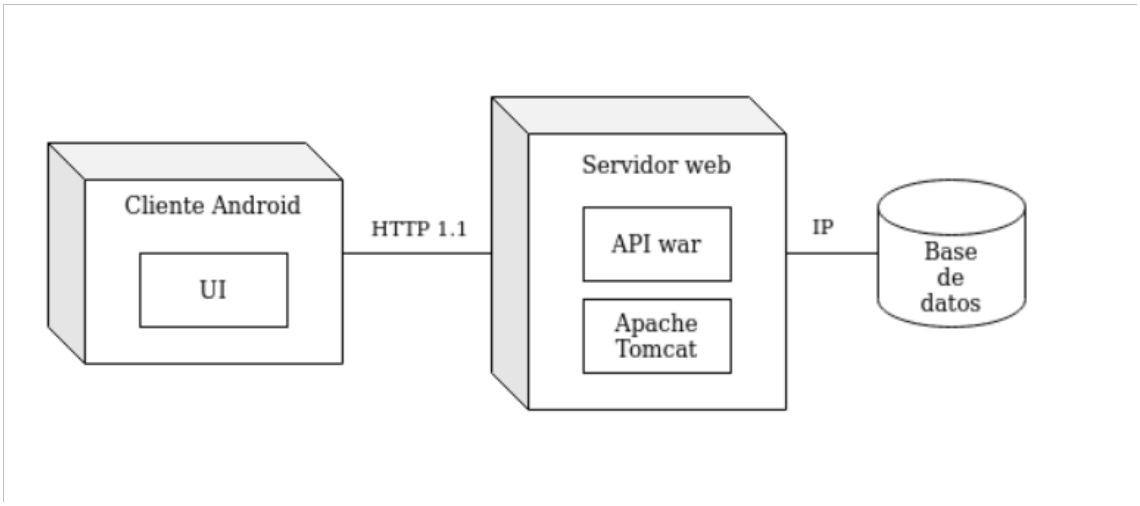
6.2. Implantación del sistema

Durante la fase de desarrollo del proyecto, tras cada iteración en la planificación de este, se ha desplegado la aplicación en un entorno controlado dentro de la empresa SOINCON. En este sistema se han probado los cambios y nuevas funcionalidades, además de realizarse las pruebas de experiencia de usuario.

Para desplegar el producto final en el cliente, al tratarse de una solución *On premises*, se necesita un entorno para su ejecución. En este caso se han solicitado dos, uno de preproducción y otro de

producción. El entorno de pre-producción tiene como finalidad la validación del funcionamiento de la aplicación por parte del cliente antes de pasar al entorno de producción, dónde se utiliza ya como aplicación operativa por parte de todos los empleados de la empresa.

Figura 6.2: Diagrama de despliegue



Ambos entornos tienen una configuración idéntica, versiones del software necesario incluidas; cada uno con su base de datos y su servidor de aplicaciones. En la Figura 6.2 se muestra un diagrama de despliegue del sistema. Se ha hecho hincapié en la configuración de red, en hacerla de la forma más parecida posible para no encontrarse con problemas en las tablas de direcciones, reglas de cortafuegos, o cualquier otra anomalía que pudiera generar un problema imprevisto. Tras la instalación se ha comprobado el correcto acceso a la aplicación en cada uno de los entornos, tanto por parte de la empresa SOINCON, como por parte de los usuarios del cliente.

Conclusiones y trabajos futuros

7.1. Conclusiones

Actualmente el desarrollo completo, esto es, la aplicación móvil, la *API REST* y una interfaz web creada por otro grupo de desarrollo de la empresa SOINCON; está instalado y funcionando en más de una veintena de clientes repartidos por la geografía Cántabra, la mayoría pertenecientes al sector industrial.

Los operarios de estas empresas ahora pueden fichar tanto desde las instalaciones, cómo desde cualquier lugar fuera de ellas dónde tengan que trabajar, con sólo tener un dispositivo a mano e independientemente de si están conectados o no a una red de datos.

Las empresas disponen de una solución que les permite consultar los fichajes de los empleados por jornada (incluyendo la localización y evidencia gráfica si lo desean), les facilita la generación de los informes de fichajes cada mes, pueden consultar los tiempos productivos de sus empleados y que les ayuda a la hora de tomar decisiones y aplicar diferentes políticas en función del análisis de los datos recogidos por el sistema.

Cabe destacar que ha dado lugar a la creación de nuevos puestos de trabajo en el área de soporte de la empresa, debido a los contratos de mantenimiento derivados de dichas instalaciones.

A nivel personal he disfrutado con el desarrollo de este proyecto, tanto por el aprendizaje que ha supuesto, cómo por la oportunidad que ha sido para aplicar lo aprendido en mis estudios de grado en la universidad. Ha supuesto un reto a nivel personal y profesional importante, del cual me alegro haber podido formar parte.

7.2. Trabajos futuros

El proyecto descrito en este documento está actualmente sumergido en un proceso de mejora y cambio. Al haber superado las expectativas de la empresa para la que se desarrolló, se ha decidido integrarla con el módulo de gestión de recursos humanos del que actualmente se dispone, conocido como *Digital People*; en su *suite* de aplicaciones, conocida como *EMI Suite*.

Gracias a esto, se pueden describir ya una serie de cambios que se desarrollarán en las próximas semanas y/o meses:

- El *backend* tendrá que integrarse con el ya existente del módulo de *Digital People*. Esto implicará una absorción del modelo de dominio, así como del contrato de la *API REST* existente. Se tendrá que refactorizar parte del código que ya no tendrá que realizar consultas entre ambos sistemas, sino que ya formará parte del mismo. Además, habrá que adecuar el contrato de la *API REST* y el acceso a los datos, ya que todos los módulos de *EMI Suite* están pensados para desplegarse como *Software as a Service (SaaS)*.

- La aplicación móvil, por otro lado, será absorbida por la aplicación ya existente de *EMI Mobile*, la cual recogerá la funcionalidad del control de asistencia tal y como se ha desarrollado en este proyecto.

- Por último, una vez se alcance el volumen de clientes necesario para hacer viable el cambio, se pretende desarrollar una plataforma de generación de informes bajo demanda con

Microsoft PowerBI, que pueda servir para ayudar a la toma de decisiones en las empresas dónde se instale.

Lista de acrónimos y abreviaturas

Android Android es un sistema operativo móvil basado en el núcleo Linux y otros software de código abierto. 3, 44

API Una API, o interfaz de programación de aplicaciones, es un conjunto de reglas que definen cómo pueden las aplicaciones o los dispositivos conectarse y comunicarse entre sí. 32, 44

API REST Una API REST es una API que cumple los principios de diseño del estilo de arquitectura REST o transferencia de estado representacional. 3, 4, 7, 12, 14, 18, 25, 27, 30, 31, 34, 39, 43, 44

Bloqueo optimista Técnica por la que si se produce un acceso concurrente a los datos, se diferencia entre dichos accesos permitiendo la operación del primero que se recibe y desechando los siguientes. Por ejemplo, si se reciben dos actualizaciones simultáneamente de la entidad Empleado, una que cambia el nombre y otra que cambia los apellidos, y la primera en llegar es la del nombre; se actualizará este dato marcando cómo inválida las siguientes que se recibieron de forma concurrente. 39, 44

End-to-End End-to-End o E2E, en el contexto de las pruebas, es la técnica te testeo que comprueba que el comportamiento del producto software es el esperado, de principio a fin. 38, 44

Framework Marco de trabajo que ofrece una estructura base para el desarrollo de proyectos con objetivos específicos. 44

GPS GPS o Global Positioning System, es un sistema de navegación que permite localizar un objeto sobre la superficie terrestre. 32, 44

HTTP Protocolo de transferencia de datos a través de la red Internet. 7, 17, 44

InnoDB Es un mecanismo de almacenamiento de datos que soporta transacciones ACID, bloqueo de registros e integridad referencial. 44

JPA JPA o Javax Persistence API, es una especificación que forma parte de JavaEE (ahora Jakarta EE) y que indica cómo se debe realizar la persistencia de objetos en Java. Es una especificación, las implementaciones son varias, pero las más conocidas son *Hibernate* y *EclipseLink*. 20, 44

JPA Criteria API Es una API para crear consultas. Con ella se pueden crear consultas de forma dinámica en tiempo de ejecución además de crear consultas seguras que puede verificar el compilador. 23, 44

JSON JSON o JavaScript Object Notation es un formato de texto utilizado en el intercambio de datos. 7, 17, 29, 44

Mock Son objetos preconfigurados y programados que simulan el comportamiento de las llamadas que se espera recibir. 36, 44

N+1 consultas Es un problema derivado del funcionamiento por defecto de Hibernate, directamente ligado a la forma de configurar el tipo de asociación entre entidades (*Lazy* o *Eager*). Tanto si no se especifica nada en algunos casos, cómo si se marca la relación cómo *Eager*, con cada consulta realizada a una entidad, Hibernate realiza otra consulta para traerse los datos de las entidades asociadas, y para cada una de estas lo mismo de forma recurrente dando lugar a las *N+1 consultas*. 39, 44

On premises Este término indica que el software se instala de manera local y se ejecuta en computadoras en las instalaciones del cliente. 41, 44

RFID Tecnología que permite la transmisión de datos a través de ondas de radio. 40, 44

TDD TDD o Test-Driven Development es una práctica de ingeniería de desarrollo de software, en la que se escriben antes los tests y después se desarrolla la funcionalidad. 36, 44

TIC Las TIC o Tecnologías de la Información y la Comunicación, son tecnologías que utilizan la informática, la microelectrónica y las telecomunicaciones para crear nuevas formas de comunicación a través de herramientas de carácter tecnológico y comunicacional, esto con el fin de facilitar la emisión, acceso y tratamiento de la información.. 3, 44

Bibliografía

[1] Ian Sommerville. "Software Engineering". En: 10.ª ed. Pearson Education Limited, 2016. Cap. 2.

[2] Scott Chacon, Ben Straub. "Pro Git". En: 2.ª ed. Apress, 2022.

[3] Tom Preston-Werner. *Versionado Semántico 2.0.0*. URL: `https://semver.org/lang/es/`.

[4] Iuliana Cosmina, Rob Harrop, Chris Schaefer, Clarence Ho. "Pro Spring 5". En: 5.ª ed. Apress, 2017.

[5] Balaji Varanasi. "Introducing Maven". En: 2.ª ed. Apress, 2019.

[6] The Apache Software Foundation. *Maven, Documentación Oficial*. 2022. URL: `https://maven.apache.org/guides/`.

[7] Jason Brittain, Ian F. Darwin. "Tomcat: The Definitive Guide". En: 2.ª ed. O'Reilly Media, Inc., 2008.

[8] The Apache Software Foundation. *Tomcat, Documentación Oficial*. 2022. URL: `https://tomcat.apache.org/tomcat-9.0-doc/index.html`.

[9] Mark Richards. "Software Architecture Patterns". En: 2.ª ed. O'Reilly Media, Inc., 2015. Cap. 1.

[10] Google Developers. *Lesson 14: Architecture Components*. 2017. URL: `https://google-developer-training.github.io/android-developer-advanced-course-concepts`.

[11] Spring. *Spring Security - Reference Documentation*. 2022. URL: `https://docs.spring.io/spring-security/reference/index.html`.

[12] Oliver Gierke, Thomas Darimont, Christoph Strobl, Mark Paluch, Jay Bryant. *Spring Data JPA - Reference Documentation*. 2022. URL: `https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html`.

[13] Reto Meier, Ian Lake. "Professional Android". En: 4.ª ed. John Wiley y Sons, Inc, 2018. Cap. 9.

[14] Google Developers. *Camera2*. 2021. URL: `https://developer.android.com/training/camera2`.

[15] Vlad Mihalcea. "High-Performance Java Persistence". En: 1.ª ed. Vlad Mihalcea, 2016. Cap. 14.